Some thoughts on teaching introductory programming and the first language dilemma

Mohammed Foughali

Some thoughts on teaching introductory programming and the first language dilemma

Mohammed Foughali
Université Paris Cité, CNRS, IRIF
Paris, France
foughali@irif.fr

ABSTRACT

There is a consensus among computer education experts and practitioners that teaching *introductory programming* is intrinsically hard. In particular, the choice of the first programming language to support learning the fundamentals of problem solving and algorithmic reasoning is a hot issue that is driving a lot of attention within the last few decades. As a side effect, the computing education community has been long divided between supporters of industrial relevance and advocates of educational benefits as the prominent grounds on which a first programming language should be elected. While the former seem to have the wind in their sails, with popular-in-industry languages such as Java, C and C++ being still widely used to teach introductory programming, the case is far from being closed. In this paper, we propose to analyze the first language choice dilemma in the light of a number of rigorous studies carried out within and outside of the computing education community. We show that, in the light of these studies, we can efficiently question our choices as educators and stimulate objective discussions toward reconciling our views regarding the first language choice. Mainly, we devise a number of criteria, all backed up with scientific findings from different communities, according to which a first language should be evaluated. Our conclusions converge toward a justified concern vis-à-vis the use of languages such as Java and C to teach introductory programming, and the pressing need for a better compromise between industrial popularity and educational advantages. To meet that need, our position gravitates around two major opinions stemming from our cross-disciplinary analysis: (i) Java, C and C++ should not be used to teach introductory programming and should rather be saved for more advanced programming courses and (ii) while the recent trend of choosing Python is justified, it is still debatable and therefore other candidates, among which we propose a couple, should also be seriously considered. Besides arguing in favor of these opinions, the primary aim of our analysis is to trigger fruitful discussions on the subject fuelled by cross-disciplinary research findings rather than personal opinions.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **Computer engineering education**.

KEYWORDS

introductory programming, first programming language, CS1 & CS2

1 INTRODUCTION

Computer science *CS* programs are proliferating across world's universities. For instance, the number of undergraduate CS programs in the US and Canada has grown by 74% in a seven-year interval [1]. Unfortunately, this expansion is contrasted with persistent difficulties for students and educators alike. In particular, *introductory programming* is noticeably hard to teach and learn [20, 23, 34, 39, 41] with rather stable, relatively high dropout rates worldwide [8, 44]. While the CS education literature is rich in debates on the "best" methodologies and languages for teaching introductory programming [9, 11, 12, 26, 40], their often divergent conclusions are hard to reconcile. As an example, Duke et al. praise Java's suitability for introductory programming [16], whereas Meyer mocks its "magic" *main* (*string*[]*args*) "which you do not understand" [32].

The aim of this paper is to set the basis for *fruitful* discussions regarding the choice of the first language used as a "support" to teach introductory programming. To that end we argue as follows. (1) Discussions on the first language choice should be made in accordance with a *final objective* (while a computer science graduate is expected to be an expert programmer [22, 41], a civil engineer is fine with basic programming skills). We focus on *introductory programming where the objective is to form expert programmers*, which we refer to simply as *IP*[1]. (2) These discussions should feed on cross-disciplinary (CS education, educational psychology, cognitive science) existing rigorous research rather than personal and anecdotal opinions. Points (1) and (2) are intimately linked: as we show in the next paragraph, a preliminary analysis of the literature points to the fact that the first language choice may seriously impede the process of acquiring expert programmer skills.

A key trait of *expert programmers EP* is that they "organise their knowledge according to functional characteristics such as the nature of the underlying algorithm rather than superficial details such as language syntax" [41], which implies the valuable benign side effect of high adaptability to new programming languages. Jenkins emphasizes this adaptability suggesting that EP simply "bring [their] knowledge to some new situation (in this case the new language)" [22, p. 45]. Unfortunately, in the last two decades, the trends in teaching IP seem to go against the EP objective. Gomes and Mendes pointed out this issue: "teachers are more concentrated on teaching a programming language and its syntactic details, instead of promoting problem solving using a programming language" [20]. More recent studies show further that the observations of Gomes and Mendes are particularly present in settings

[1] IP can be viewed as CS1 + CS2, but we prefer to clearly define what it is in order to avoid the possible confusion the latter acronyms may entail [21].

where the language used to teach IP is chosen primarily for its industrial popularity [25, 43]. In particular, Koulouri et al. show in an elegant study that teaching "some [popular-in-industry] programming languages [with a focus on Java]" may drown beginners into syntax issues and "distract them from learning basic programming concepts, which may, in turn, have a lasting impact on students' confidence" [25]. There appears therefore to be a discrepancy between the choice of languages such as Java and C (mainly for their industrial merit) in IP courses, still popular as of today, and the consensus that teaching IP (the objective of which is, we recall, forming EP) must focus more on algorithmic reasoning than on programming languages technical details [20, 22, 25]. This situation is, to say the least, worrying, given that several studies confirm that educators are clearly shifting from the classical debate of industrial relevance *vs.* educational benefits toward settling for the former as the most important — even the only — ground for selecting the first language (Sect. 4). There is therefore an urgent need to reconcile our views on the first language dilemma somewhere between educational fitness and industrial relevance. Even with the recent trend of switching to Python, this urgent need is far from being met, as reflected by the persistence of Java/C/C++ in IP courses today (Sect. 4) and the educational pitfalls of Python (Sect. 3, 4). Therefore, in order to question our choices as educators, we need to establish clear, evidence-based criteria on which electing a language for teaching IP should be based.

This paper identifies some of these criteria and discusses whether the current choices w.r.t. the first language should be reconsidered. While we (i) make a clear case against the use of Java, C and C++ to teach IP and suggest saving them for later years (when students are no longer novices) and (ii) put forward three alternative language candidates (including Python), our main objective is to trigger constructive discussions on the first-language choice dilemma based on cross-disciplinary scientific findings. Note that the conclusions of this paper rely on a *procedural-first approach*, without any attempt to minimize the debate on the first-paradigm dilemma [10, 37] (more in Sect. 4).

Following the importance and the dependency between points (1) and (2) above, the paper is organized as follows. First, we provide a context of a typical IP course for which a programming language should be chosen to serve as a "support" (i.e. to implement the algorithms). In particular, we discuss the educational choices made in the course, in light of various research findings, in order to be in line with the EP objective, regardless of the support language (Sect.2). Second, we devise some criteria, also backed up with existing research, to elect a first language to support the course presented previously (Sect. 3). We show that languages such as C, C++ and Java, still as of today belonging to mainstream languages used in IP courses[2], score poorly when confronted with these criteria. We then conclude with Sect. 4 as we propose, besides Python, already considered by many educators, two other candidates, namely Ada and Kotlin, to possibly replace Java/C/C++. We show that (i) Python is not necessarily the ideal undebatable replacement and (ii) languages like Ada, and perhaps Kotlin, should not be discarded a priori, and conclude with open questions regarding the possible directions of future discussions where further candidates may also be considered.

[2]References on the popularity of these languages in IP courses are further detailed in Sect. 4

2 IP COURSE

2.1 Targeted public

The course we present is applicable to any IP (i.e. any introductory programming, first-year university course where the final objective is to form EP, Sect. 1). It was originally implemented and taught by educators (including ourselves) in the French Engineering School (*Ecole d'Ingénieurs*) INSA Toulouse[3], with 21 lectures and assignment sessions (1.25 hours each) and 11 lab sessions (2.75 hours each) overall. French Engineering Schools may be viewed as "classical" universities except that, essentially, *Science, technology, engineering, and mathematics (STEM)* programs are taught within a denser curriculum. In the first two years, students are taught, besides IP, a number of STEM subjects (including various physics, chemistry, and mathematics disciplines) after which they choose an orientation for the remaining three years. Many orientations are within CS and Computer Engineering CE, such as *Distributed Systems* and *Artificial Intelligence*, where at least two more programming languages are taught. CS and CE graduates are expected to easily adapt to new programming languages in order to be highly employable, hence the EP final objective.

2.2 Content

The course frames programming as a *communication problem*. We have, at one end, a human speaking natural language (e.g. English or French) and, at the other end, the computer "speaking" binary. The problem of programming then consists in finding a way for the human to communicate with the computer and get it do what he/she wants i.e. execute the *solution* to the *problem* at hand. Since the gap between the languages spoken at each end is huge, *programming languages* serve as an *interface* (in the sense of a "compromise") between the human and the computer. Still, a programming language is drastically different than natural language, so we need another "intermediary interface": *high-level solutions*.

Fig. 1, taken from our teaching slides, illustrates the programming problem as introduced in the course. Connecting the two ends of the communication and the interfaces forms a chain of programming "phases": (i) specifying solutions (the *specification* phase), (ii) implementing them as programs using a programming language (the *implementation* phase) and (iii) compiling such programs into binaries (the *compilation* phase)[4].

The course explains afterwards that compilers are ready-to-use generators: the compilation phase does not need any intervention from the students (besides interpreting its results). It summarizes accordingly that programming is a two-phase activity: at the specification phase, *what we want the computer to do* transforms from natural language to abstract mathematical solutions, whereas the latter become programs at the implementation phase. The course emphasizes the importance of the specification phase as mastering

[3]https://www.insa-toulouse.fr (in French).

[4]It is perhaps noteworthy to mention that, in a purely historical context, Friedman presented an organization structurally similar to Fig. 1 using a *ladder* [19] (yet different in content, as it had nothing to do with teaching).

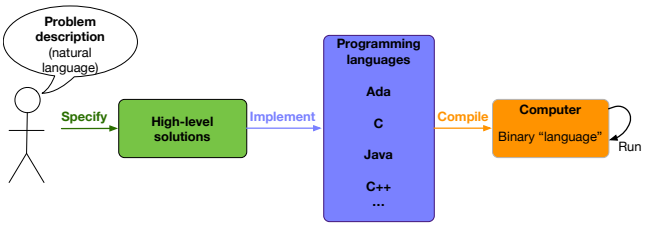

Figure 1: The programming problem.

it will make it easier for students to succeed in the different programming languages courses in the upcoming years. It is important to note that the organization in Fig. 1 is paradigm independent . While this course follows a procedural-first approach, where, in later slides, the green box content is replaced with "algorithms", the same organization can be used within an object-first approach (by simply replacing "high-level solutions" with e.g. "diagrams & algorithms" in later slides).

The course then focuses on the specification problem: *how do I specify an abstract solution to a problem described in natural-language?* In a procedural setting, the specification phase is equivalent to writing algorithms. Based on Fig. 1, to program is to communicate with the computer, thus solutions should be written *for* the computer. Therefore, an algorithm should provide a solution that a high-level fictive computer would be able to execute. This fictive computer has different levels of "tolerance" toward what humans write. The tolerance-zero level is conceptual: this involves e.g. a sequence of instructions that are both elementary (adding two numbers is elementary, factorial is not) and unambiguous ("do this" is clear, "do this or that" is not). The maximum-tolerance level is syntactical, e.g. ending an instruction with either a line-break or a semicolon is okay as long as it is consistent throughout the algorithm. Thus, the course encourages the students to separate more important aspects, that is problem solving (how to solve a problem thinking like the fictive computer, e.g. in terms of a sequence of elementary unambiguous instructions) from the less important ones, that is syntactical details (e.g. what symbol should I use to end my instruction?)

On this basis, the remaining course sessions (lectures and assignments) cover various aspects of algorithms (e.g. types, variables, instructions, loops and subroutines) and data structures (e.g. records and arrays). IP is limited to static variables (dynamic variables and pointers are covered in the second year). Implementation is introduced gradually as the students encode their algorithms using a support language. Since algorithms and data structures are classic in IP, we do not go into their details. Rather, we emphasize some choices made in the course and explain their motivations.

2.2.1 Algorithmic notations. McIver and Conway referred to seminal works from educational psychology and cognitive science to argue that learning programming can be negatively affected by unlearning existing parts of the students cognitive structures [31] (for instance, unlearning that = means *equal to*, a knowledge that first-year students hold for over 15 years). Teaching algorithms to a beginner should connect to his/her existing cognitive structures and involve as little unlearning as possible, preferably no unlearning at all (see more on the effects of unlearning in Sect. 3.2.1). Thus, = remains the notation for equality, whereas we use ← for assignment (read "transfer what is on the right into what is on the left" [31]). Similarly, we use either universal symbols or their English meanings for boolean operators, e.g. ∧ and ∨ (or simply *and* and *or*) . Besides being "anti-unlearning" and promoting simplicity (more on the importance of syntax simplicity at the implementation level is given in Sect. 3.2.2), these choices serve the purpose of building a "universal" specification basis, independent as much as possible from programming languages. Students will thus always find their "pivot" when moving to a new programming language, as they implement their abstract solutions in the language (mapping the universal symbols to their language-specific counterpart).

2.2.2 Variables. Variable-related misconceptions are a major source of students difficulties in programming [33]. These misconceptions can form as early as the abstract introduction of variables in IP. Clancy, for instance, shows that using a box to represent a variable (a metaphor still commonly employed by teachers [39]) messes with the mental model of students as it makes them, for instance, think that a variable may hold a number of things [13]. We suggest using the *labelized erasable slates* metaphor: a variable is an erasable slate with a label indicating its name and type. Thus, declaring e.g. an integer variable x creates a slate, labelized with the variable name x and type *integer*, with an arbitrary integer on it, which students may erase to write a different value instead (assignments, including initialization). The label allows to find the variable (the name) and prevents writing anything but integers (the type).

2.2.3 Loops. Another widespread misconception is viewing the *while* and the *for* loops as identical, therefore preferring one over the other instead of comprehending that each is destined to solve a different problem [39]. While Qian et al. link this misconception to factors such as "which loop construct the student learned first" [39], we show in Sect. 3.2.3 that the programming language chosen for IP is another important factor. In the course we present, comprehending these loop constructs regardless of the language is important. Finding an integer in an array is a *while* problem (browse the array *as long as* the value is not found or there are more cells to browse) whereas finding the smallest integer in an array is a *for* problem (browse the whole array in any case, i.e. *for* each index in the array range). Understanding the differences at this stage minimizes the chances of students to confuse both constructs (Sect. 3.2.3).

2.2.4 Subroutines. Though distinguishing subroutines with and without a return value may sound trivial, giving it little attention at the specification phase has serious consequences at the implementation phase (Sect. 3.2.4). The course uses the terminology *procedure* and *function* to denote a subroutine without and with a return value, respectively. In addition, the course advocates the "good practices" of writing subroutines as early as the specification phase. These practices include the well-known "friends" of reusability (e.g. simple, hardcoding-free subroutines), but also other guidelines such as "a return values must not be discarded" to help prevent common, hard-to-catch code bugs (Sect. 3.2.4).

2.2.5 Analysis. Algorithms must be correct, in order to implement correct code. Many students make mistakes (e.g. infinite *while* loops)

due to violating the underlying logical properties of different algorithmic concepts. Therefore, the course promotes logical reasoning as it guides students toward deriving formal rules that help them write correct algorithms. For example, let V (resp. I) be the set of variables tested in the condition (resp. the set of instructions within the body) of a *while* loop, and $SI : I \mapsto \mathcal{P}(V)$ (where $\mathcal{P}(V)$ is the powerset of V) the function returning the subset of V modified by an instruction $i \in I$. The course drives students' attention to the fact that if $\forall i \in I : SI(i) = \emptyset$ then the *while* loop is either never executed or infinite (and is thus, in the general case, misconceived). Moreover, the course familiarizes students with program verification. In particular, students learn how to use preconditions, postconditions and invariants to verify their algorithms. At the implementation phase, students should be able to concretize their acquired knowledge on program verification in lab sessions (Sect. 3.2.5).

3 FIRST LANGUAGE CHOICE

In this section, we reflect on the first language choice dilemma in the context of the IP generic course introduced in Sect. 2. Like it was the case for the choices made in the course, we evaluate the support language according to several criteria in the light of rigorous, cross-disciplinary research findings. The criteria belong to the two traditional categories considered by computing education experts when electing a language to support teaching IP: industrial relevance (Sect. 3.1) and educational benefits (Sect. 3.2). In particular, we derive five important aspects under the educational benefits category based on previous research.

We analyse, under all criteria, six programming languages: Java, C, C++, Python, Ada and Kotlin. These candidates are chosen in compliance with our observations, backed up with the literature, introduced in Sect. 1. Indeed, we evaluate Java, C and C++ to underpin the concerns raised in [20, 25, 43] pertaining to the fact that using these languages in IP, based on their sole industrial merit, threatens attaining its EP final objective. Python is analyzed as it has been, in the last decade, the de facto choice of educators that wish to reconsider using Java/C/C++ (more in Sect. 4). Ada and Kotlin are also scrutinized in order to discuss whether they could be viable candidates to replace Java/C/C++ in IP so the latter may be taught in later years in the curriculum, when the students are no longer total beginners.

3.1 Industrial relevance

In this category, the weak candidates are, without a doubt, Ada and, to a lesser extent, Kotlin. Indeed, Java, C and C++ are known for being traditionally used in industry throughout the last decades, while Python is galloping when it comes to industrial popularity. For instance, the most recent survey of the Indeed job-application platform places these four languages among the top-five most demanded languages in industry [5]. (Another abundantly detailed survey with similar findings is available from StackOverflow [6]). However, to be fair with the weak cadidates, Ada's and Kotlin's lack of popularity should not be mistaken for a total absence in industry as it is the case for languages specifically tailored for education such as Eiffel and Pascal. Industrial relevance of both Ada and Kotlin is discussed next.

Ada has a long-term ongoing history with industrial critical domains worldwide[7]. For instance, the Ariane 4/5/6 rockets software was written in Ada, and the Systerel company [8], a major actor of critical computing in France, regularly hires Ada experts. Moreover, the recent ESROCOS and ADE projects [4, 35], funded by the European Space Agency, heavily relied on Ada (within the TASTE toolbox [38]).

As for Kotlin, and since being announced as the preferred language for developing Android applications by Google in 2017, it gained significant popularity as other major companies such as Uber and Pinterest adopted it for purposes that go sometimes beyond mobile applications development [36].

3.2 Educational benefits

We compare the six languages within five non-exhaustive criteria backed up with relatively recent scientific findings.

3.2.1 Anti-unlearning syntax. As explained in Sect. 2.2.1, unlearning existing parts of cognitive structures has negative effects on beginners. The equal/assignment confusion is a famous unlearning side effect where a beginner would write e.g. $if\,(x = 5)\{\dots\}$ (instead of $if\,(x == 5)\{\dots\}$) in C, C++, Java, Python or Kotlin thinking that the body of *if* will be executed only if x is equal to 5 (while, in reality, 5 is assigned to x silently as the condition of *if* becomes a tautology). This issue seems to receive much less attention than it should, as Stefik and Siebert showed in a rigorous study [43]. In particular, the authors noticed that the double-equal sign was used correctly by beginners as little as any random sign such as the *Bang* (!) which the authors invented for their experiments (less than 2% used either sign as they should have). Stefik and Siebert concluded that "We understand all too well why many language designers historically made the choice of ==, but the impact on novices is clear". Moreover, in a more recent thorough study (data collected over more than 37 million compilations), Altadmri and Brown showed that the equal/assignment confusion ranked high in terms of both frequency (4^{th}) and difficulty to fix (6^{th}) among the 18 most common errors committed by beginners [3]. Another example of unlearning side effects is the 0 index in C, C++, Java, Python and Kotlin, frequently at the origin of the famous *off-by-one error*: students need to unlearn *first* being simply 1^{st} and replace it with *first* being 0^{th}. In general, the chosen language for IP must involve as little unlearning as possible, a criterion amply met by Ada. Contrary to the other languages under scrutiny, Ada has an anti-unlearning syntax: equal is =, whereas := (inherited from the educational language Pascal) is used for assignments, and the keywords *'First* and *'Last* are provided for arrays boundaries.

In sum, only Ada meets this criterion.

3.2.2 Simple syntax. The chosen support language should have simple syntax, avoiding cryptic symbols [28] and "magic formulae" [32]. C, C++ and Java use cryptic syntax (e.g. && for $\wedge$ and || for $\vee$), that does not intuitively connect to universal mathematical notations (Sect. 2.2.1). Moreover, writing the first program in

[5] https://uk.indeed.com/career-advice/career-development/coding-languages
[6] https://survey.stackoverflow.co/2023/#most-popular-technologies-language

[7] See e.g. [27] and https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html
[8] https://www.systerel.fr/en/

any of these languages requires beginners to use a "magic formula" which they will likely not understand before the next year, such as *main* (*string* [] *args*) in Java. Besides, these languages use semicolons the forgetting and misplacement of which are very common errors among beginners (in the top fifteen of most frequent errors according to Altadmri and and Brown's study [3]). Both Ada and Python enjoy simple and intuitive syntax: for instance, the English keywords *and* and *or* are used for the obvious corresponding logical operators, and writing the first program is magic formula free (e.g. in Ada the main program is simply a parameterless procedure). However, contrary to Python and Kotlin, Ada uses semicolons in a similar fashion as Java, C and C++. As for Kotlin, though it shares cryptic syntax with Java/C/C++, a program can be written easily in a magic-formula-free manner, and the language shares the easy use of line-breaks to end instructions with Python, with the extra simplicity of insensitivity to indentations.

In brief, Ada, Python and, to a lesser extent, Kotlin partially meet this. Java, C and C++ are the weak candidates.

3.2.3 Clear concepts of loop constructs. The support programming language should concretize the understanding of the conceptual differences between a *while* and a *for* loop, acquired at the specification phase (Sect. 2.2.1). This is the case of Ada, providing a clear, unambiguous syntax for both loop constructs. The syntax of *while* is similar to that of C, but the *for* loop construct is different: *for x in r do ...* where x is a variable and r is a range, thus the instructions within the body of *for* are repeated for the whole range r. In C/C++, the *for* loop simply does not exist, although a *for* construct is provided. Actually, the latter is a *while* construct written in a more compact way: *for*(*inst1*; *cond*; *inst2*) is a *while* loop preceded with instruction *inst1*, conditioned with the Boolean expression *cond* and incorporating *inst2* as the last instruction in its body. This is surprising, and remarkably poor from a conceptual point of view: it promotes the misconception of both loops being similar, thus pushing the students to compare them [39]. Eventually, beginners that will deem *for* better than *while* will end up writing a *for* loop with a *break* (or even worse, a *return*) statement to solve a typical *while*-loop problem. In Java, things are even worse: the *for* loop (which is actually a *while* loop) is kept from C, and a *foreach* loop construct is introduced to implement a true *for* loop, but it is called also *for* (one can only imagine the confusion this creates in a novice student's mind). Python and Kotlin, on the other hand, share with Ada a clear and nice separation between both loop constructs.

To conclude, Ada, Python and Kotlin are excellent candidates w.r.t. this criterion, which neither Java, C or C++ meet.

3.2.4 Clear and rigorous concepts of subroutines. The support language should concretize the distinction between subroutines with and without return values (learned at the specification phase, Sect. 2.2.4) in a rigorous manner. In Ada, the syntactical distinction is explicit: *procedure* (no return value) and *function* (one return value). In C, C++, Java, Python and Kotlin, it is the return type of the subroutine that makes the difference (though less intuitive in Python, where types are implicit). Syntactically, this is not a real plus of Ada. However, Ada's rigorous separation between procedures and functions becomes clearly advantageous at compile-time. Let us illustrate with an example. Listing 1 shows the same erroneous student code in Ada (right, where x has been declared integer earlier in the program) and Python (left).

```
1  x = 5
2  print("the factorial of", x)
3  fact(x)
4  print("is", x)
```

```
1  x:= 5;
2  Put("the factorial of" &
         Integer'Image(x));
3  fact(x);
4  Put("is" & Integer'Image(x));
```

Listing 1: Discarding/ignoring return values of non-void functions in Python (left) and Ada (right)

The student is trying to display x, update it with its factorial $fact(x)$, then display x again. However, he/she forgot to assign the return value of $fact(x)$ to x (line 3). The Python code compiles (as its equivalent in C, C++, Java or Kotlin would), while it does not in Ada with the compiler displaying "cannot use function *fact* in a procedure call" pointing to line 3. This is a rather dummy example, but it highlights a remarkably serious mistake (known as *discarding/ignoring return values of non-void functions*), ranked 1^{st} in terms of *average time to fix* ($\geq$ *1000* seconds) in Altadmri and Brown's study [3] (referenced earlier in Sect. 3.2.1, 3.2.2). Such high average would be drastically reduced with Ada, as the error is caught by the compiler. While discarding return values in the remaining language candidates looks like a trivial matter (easily fixable at the compiler level), the historical choices made in these languages makes it tricky in practice. For instance, C++17 introduced the "no discard" attribute to explicitly denote that a method may not see its return value discarded. Evidently, this is an additional burden for a beginner student.

To summarize, Ada is the only candidate that satisfies this criterion.

3.2.5 Verification support. Finally, a good language for IP is also a language that promotes writing correct code [28] (which connects to specifying correct algorithms, Sect. 2.2.5). Ada is known for its strong typing and strict compiler, both promoting writing correct programs. One example on strict compiler benefits is the *discarding/ignoring return values* error, which only Ada catches at compile-time (Sect. 3.2.4). Let us now give an example on how strong typing can help catch errors *at runtime*. Consider the Kaprekar routine, where students store, in an array A, the digits of a number N, then use them to perform some computations. A hard-to-catch mistake is when e.g. a two-digit number is put in one of A's cells. Ada's strong *range subtyping* can help. It suffices to write *subtype Digit is Natural range 0..9;* then define A as an array of *Digit*, then, at runtime, if any instruction attempts to put a value outside of *0..9* in a cell of A, the execution stops with the message "constraint check failed", pointing to the culprit instruction. In contrast, range subtyping does not exist per se in C, C++, Java or Kotlin (despite some online projects that claim to implement it), which would entail, for e.g. the Kaprekar example, longer classic debugging through inserting *print/printf* statements. In Python, things are more complicated as the language uses a sort of implicit typing.

Let us consider another example where a student is supposed to implement a function that computes the factorial $fact(n)$ of a number n. Here, the *fact* function should implement its classical

mathematical counterpart *fact* : $\mathbb{N} \mapsto \mathbb{N}$, that is *fact* is only defined for positive integers, and always returns a positive integer. In Ada, this is easily feasible through defining both the only parameter of the function and its return value as naturals:

```
function fact (n : Natural) return Natural
```

A direct advantage of strong subtyping here is avoiding, by design, ill-typed instructions (for instance calling the function above with a negative integer) and return values outside of the function's range. But strong subtyping benefits go beyond this direct advantage. In this example, it allows for a rigorous mapping of a function, as a mathematical object at the specification phase, to a function, as a programming concept. To explain this, let us see how the C/C++ header of the same "function" would look like:

```
int fact (int n)
```

But, strictly speaking, *fact* above is not a function to begin with (simply because it is not left total), contrary to the Ada implementation. For a novice programmer, using a language that allows precise mapping of high-level solutions (algorithms) and their mathematical foundations (e.g. functions) to programs promotes writing correct code. This is not the case in the other candidate languages, especially in Python because of the absence of explicit typing.

Strong (sub)typing put aside, Ada has solid foundations in program verification: its SPARK subset/toolset [6]. C, C++, Java, and Kotlin verification support is more or less comparable to SPARK (e.g. the state-of-the-art KeY theorem prover for JavaCard [2]). As for Python, Nagini has been recently proposed [17]. However, it requires a strictly typed version of Python, which clashes with the "simplicity" induced by the implicit typing philosophy of Python.

In fewer words, Ada is an ideal candidate under this criterion. C, C++, Java and Kotlin are more difficult to use to write correct code (due to typing/subtyping limitations), but they may be still fit thanks to their verification support. Python is the weak candidate due to the absence of explicit typing.

4 DISCUSSION & CONCLUSION

When it comes to choosing a programming language for IP, the CS education community has been long polarized with advocates of industry relevance opposed to those of educational benefits, and the former seem to have the wind in their sails. Many studies show that teachers first criterion to choose a language for IP is industry relevance, even before pedagogical suitability [14, 15, 28]. As a consequence, languages such as Java, C and C++ are still very popular in IP courses today, despite the recent ascension of Python. For example, and according to the most recent rigorous surveys that we could find on the most popular languages used to teach IP, Java and C appear among the top-three of such languages in the UK and Australia [30] and Ireland [7], and C is overwhelmingly dominant in IP courses in Portugal (according to a 2019-2020 data shared by public Portuguese universities and reported in [42]) and in Greece [5]. This is a sad state of affairs, as such choices are more dangerous than one may think (Sect. 1, Sect. 3). It is thus perhaps about time to revisit the "industry-relevance-first" choice, and reserve these languages for teaching programming at more advanced levels (second-year students and above). With the rise of Python, it is now clearer than ever that to avoid a never-ending war (in which "the loudest professors always win" [24]), the right posture to adopt must be more flexible: both educational and industry relevance are important [15]. Perhaps, Python is slowly conciliating the two extremes as it is both gaining popularity in industry and regarded as education friendly [29], but the case is not yet closed. First, besides its advantages that our analysis confirms to some extent, Python is still far from being the ideal, undebatable replacement as we have shown through the same analysis. Second, the fact that Java/C/C++ are still widely used in IP courses, as detailed above, attests to the lack of unanimity regarding the trend of switching to Python.

Our category-based analysis on educational fitness (Sect. 3.2) shows a clear advantage of Ada, followed by Python, and to a lesser extent Kotlin, with C, C++ and Java lagging behind. The analysis is not exhaustive, but still covers fundamental criteria of educational fitness, long-established by educational psychologists (e.g. categories one, Sect. 3.2.1 and two, Sect. 3.2.2) and/or crucial to avoid the underlying misconceptions of the most tenacious mistakes of first-graders reported in rigorous research (e.g. category four, Sect. 3.2.4). Ada's educational superiority over Python and Kotlin, mainly stemming from its anti-unlearning syntax, strong typing and strict compiler, is contrasted with an industrial relevance that seems to be tied to critical applications (Sect. 3.1), making it the less desirable language form an industry-relevance angle especially with the exploding popularity of Python (and the rising one of Kotlin). While our positive observations on Python are corroborated by Koulouri et al.'s rigorous analysis [25], we did not find any work that includes Ada as a first language candidate (except for a couple of over 30-year-old publications [18, 45]), although the language is used, at least in a number of French universities, to support IP courses. As for Kotlin, its timid to non-existent appearance as an IP candidate in previous research is probably due to it being the youngest among our candidates. Yet, according to our criteria, it is still noticeably behind Ada (educational-fitness-wise) and Python (industry-relevance-wise). Perhaps, we could sum up accordingly the takeaway of our analysis in the three following conclusions: (1) Java/C/C++ should be avoided in IP courses, (2) the trend of using Python, while somewhat justified, is clearly debatable and (3) Ada, and perhaps Kotlin, should receive more attention as potential candidates (the former might have been discarded too quickly, and it might be about time to start considering the latter).

At this stage, we would also like to (re)emphasize the context in which our analysis remains valid. As explained in Sect. 1, the objectives of the IP course are a crucial parameter toward electing a first language. In other words, the way we, educators, "split" learning objectives between IP and higher-level programming courses is a determining factor for choosing the first language. Our analysis is carried out with, in mind, a broadly brushed final objective of forming expert programmers (Sect. 1, Sect. 2), with the hope that its results remain in line with the smaller objectives resulting from the decomposition of the eventual one. Similarly, and in a closely connexe scope, the choice of the first paradigm is equally important and may greatly influence the usefulness of our analysis. This is, however, out of the scope of this paper where we assume a procedural-first approach, without any intention, nevertheless, to minimize the importance of the first-paradigm dilemma.

As explained in the introduction, this paper is not about personal opinions, it rather seeks triggering discussions based on scientific evidence. The three overall conclusions above are thus meant to

set some flexible bases for such discussions. We wrap up therefore with the following questions to serve as stimulating headlines:

- Are we ready, as educators, to keep taking the risk of thinking in terms of "industrial relevance first" and thus using languages such as Java, C and C++ to teach IP knowing their serious educational drawbacks?
- If no, how do we define a balance between industrial popularity and educational advantages?
- Does Python realize this balance despite its implicit typing, porous compiler and unlearning-friendly syntax?
- Is Ada a viable candidate even with a rather poor industrial relevance?
- Could Kotlin be envisaged as a serious candidate regardless of its unfriendly syntax and yet-to-be-proven industrial prosperity? What about other modern languages like Rust?

ACKNOWLEDGMENTS

Special thanks to Didier Le Botlan (INSA Toulouse) who, thanks to his substantial efforts, both on the pedagogical and technical fronts, as well as the constructive role he played in our passionate debates (mostly ending in agreements), greatly inspired this paper. We also thank Aldric Degorre (Université Paris Cité/IRIF) for his valuable insights on Kotlin.

REFERENCES

[1] 2018. *Assessing and responding to the growth of computer science undergraduate enrollments.* National Academies Press.
[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, et al. 2005. The key tool. *Software & Systems Modeling* 4, 1 (2005), 32–54.
[3] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *SIGCSE Technical Symposium*. 522–527.
[4] Miguel Muñoz Arancón, Malte Wirkus, Killian Hoeflinger, Nikolaos Tsiogkas, Saddek Bensalem, Olli Rantanen, Daniel Silveira, et al. 2019. ESROCOS: Development and Validation of a Space Robotics Framework. In *Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*.
[5] Nikolaos Avouris. 2018. Introduction to computing: a survey of courses in Greek higher education institutions. In *Pan-Hellenic Conference on Informatics*. 64–69.
[6] John Gilbert Presslie Barnes. 2003. *High integrity software: the SPARK approach to safety and security*. Pearson Education.
[7] Brett A Becker. 2019. A survey of introductory programming courses in Ireland. In *Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. 58–64.
[8] Jens Bennedsen and Michael E Caspersen. 2019. Failure rates in introductory programming: 12 years later. *ACM Inroads* 10, 2 (2019), 30–36.
[9] László Böszörményi. 1998. Why Java is not my favorite first-course language. *Software-Concepts & Tools* 19, 3 (1998), 141–145.
[10] Susan S Brilliant and Timothy R Wiseman. 1996. The first programming paradigm and language dilemma. In *SIGCSE technical symposium on Computer science education*. 338–342.
[11] Neil CC Brown and Greg Wilson. 2018. Ten quick tips for teaching programming. *PLoS computational biology* 14, 4 (2018), e1006023.
[12] Shyamal Suhana Chandra and Kailash Chandra. 2005. A comparison of Java and C#. *Journal of Computing Sciences in Colleges* 20, 3 (2005), 238–254.
[13] Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. *Computer science education research* (2004), 85–100.
[14] Stephen Davies, Jennifer A Polack-Wahl, and Karen Anewalt. 2011. A snapshot of current practices in teaching the introductory programming sequence. In *SIGCSE Technical Symposium*. ACM, 625–630.
[15] Michael De Raadt, Richard Watson, and Mark Toleman. 2003. Language tug-of-war: industry demand and academic choice. In *Australasian Computing Education Conference*. Australian Computer Society Inc., 137–142.
[16] Roger Duke, Eric Salzman, Jay Burmeister, Josiah Poon, and Leesa Murray. 2000. Teaching programming to beginners - choosing the language is just the first step. In *Australasian conference on Computing education*. ACM, 79–86.
[17] Marco Eilers and Peter Müller. 2018. Nagini: a static verifier for Python. In *International Conference on Computer Aided Verification (CAV)*. Springer, 596–603.
[18] Thomas S Frank and James F Smith. 1990. Ada as a CS1-CS2 language. *ACM SIGCSE Bulletin* 22, 2 (1990), 47–51.
[19] Linda Weiser Friedman. 1992. From Babbage to Babel and beyond: A brief history of programming languages. *Computer Languages* 17, 1 (1992), 1–17.
[20] Anabela Gomes and António José Mendes. 2007. Learning to program - difficulties and solutions. In *International Conference on Engineering Education (ICEE)*.
[21] Matthew Hertz. 2010. What do "CS1" and "CS2" mean? Investigating differences in the early courses. In *SIGCSE Technical Symposium*. 199–203.
[22] Tony Jenkins. 2001. The motivation of students of programming. Master's Thesis.
[23] Tony Jenkins. 2002. On the difficulty of learning to program. In *Ann. Conf. of LTSN Centre for Inf. and Comp. Sc.*, Vol. 4. Citeseer, 53–58.
[24] Tony Jenkins. 2004. The first language - a case for Python? *Innovation in Teaching and Learning in Information and Computer Sciences* 3 (2004), 1–9. Issue 2.
[25] Theodora Koulouri, Stanislao Lauria, and Robert D Macredie. 2014. Teaching introductory programming: A quantitative evaluation of different approaches. *ACM Transactions on Computing Education* 14, 4 (2014), 1–28.
[26] Scott Leutenegger and Jeffrey Edgington. 2007. A games first approach to teaching introductory programming. In *SiGCSE Bull.*, Vol. 39. 115–118.
[27] Kenneth Magel. 2017. Revisiting the Impact of the Ada Programming Language. *Computer* 50, 9 (2017), 10–11.
[28] Linda Mannila and Michael de Raadt. 2006. An objective comparison of languages for teaching introductory programming. In *Koli Calling International Conference on Computing Education Research (Koli Calling)*. 32–37.
[29] Raina Mason, Graham Cooper, and Michael de Raadt. 2012. Trends in introductory programming courses in Australian universities: languages, environments and pedagogy. In *Australasian Computing Education Conference*. 33–42.
[30] Raina Mason, Tom Crick, James H Davenport, and Ellen Murphy. 2018. Language choice in introductory programming courses at Australasian and UK universities. In *SIGCSE Technical Symposium*. 852–857.
[31] Linda McIver and Damian Conway. 1996. Seven deadly sins of introductory programming language design. In *International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*. IEEE, 309–316.
[32] Bertrand Meyer. 2003. The outside-in method of teaching introductory programming. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 66–78.
[33] Iain Milne and Glenn Rowe. 2002. Difficulties in learning and teaching programming. *Education and Information technologies* 7, 1 (2002), 55–66.
[34] IT Chan Mow. 2008. Issues and difficulties in teaching novice computer programming. In *Innovative techniques in instruction technology, e-learning, e-assessment, and education*. Springer, 199–204.
[35] Jorge Ocón, Iulia Dragomir, Andrew Coles, A Green, L Kunze, R Marc, CJ Perez, et al. 2020. ADE: Autonomous DEcision making in very long traverses. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*. Lunar and Planetary Institute.
[36] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. 2020. On the adoption of kotlin on android development: A triangulation study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 206–216.
[37] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *Working group reports on ITiCSE on Innovation and technology in computer science education* (2007), 204–223.
[38] Maxime Perrotin, Eric Conquet, Julien Delange, AndrŰ Schiele, and Thanassis Tsiodras. 2011. TASTE: A real-time software engineering tool-chain overview, status, and future. In *International SDL Forum*. Springer, 26–37.
[39] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education* 18, 1 (2017), 1–24.
[40] R Rahul, Ashwin Whitchurch, and Madhav Rao. 2014. An open source graphical robot programming environment in introductory programming curriculum for undergraduates. In *International Conference on MOOC, Innovation and Technology in Education (MITE)*. IEEE, 96–100.
[41] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.
[42] Sónia Rolland Sobral. 2021. The old question: which programming language should we choose to teach to program?. In *International Conference on Advances in Digital Science*. Springer, 351–364.
[43] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Trans. on Comp. Ed.* 13, 4 (2013), 1–40.
[44] Christopher Watson and Frederick WB Li. 2014. Failure rates in introductory programming revisited. In *Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. 39–44.
[45] Leon E Winslow and Joseph E Lang. 1989. Ada in CS1. In *SIGCSE Technical Symposium*. 209–212.