A necessity-driven ride on the abstraction rollercoaster of CS1 programming

Marco Sbaraglia, Michael Lodi, Simone Martini

A necessity-driven ride on the abstraction rollercoaster of CS1 programming☆

Marco SBARAGLIA[1,3,*], Michael LODI[1,2,3], Simone MARTINI[1,2,3]

[1] *Dipartimento di Informatica - Scienza e Ingegneria, Alma Mater Studiorum- Università di Bologna, Bologna, Italy*
[2] *Inria Sophia Antipolis-Bologna, Valbonne, France*
[3] *Laboratorio Nazionale CINI Informatica e Scuola, Italy*
e-mail: marco.sbaraglia@unibo.it, michael.lodi@unibo.it, simone.martini@unibo.it

Abstract. Introductory programming courses (CS1) are difficult for novices. Inspired by *Problem solving followed by instruction* and *Productive Failure* approaches, we define an original "necessity-driven" learning design. Students are put in an apparently well-known situation, but this time they miss an essential ingredient (the target concept) to solve the problem. Then, struggling to solve it, they experience the necessity of that concept. A direct instruction phase follows. Finally, students return to the problem with the necessary knowledge to solve it. In a typical CS1 learning path, we recognise a challenging "rollercoaster of abstraction". We provide examples of learning sequences designed with our approach to support students when the abstraction changes (both upward and downward) inside the programming language, for example, when a new construct (and the related syntactical, conceptual, and strategic knowledge) is introduced. Also, we discuss the benefits of our design in light of Informatics education literature.

Key words: Abstraction, Abstraction rollercoaster, Necessity, Necessity mechanism, Necessity learning design, Learning design, CS1 learning design, Productive Failure, Problem Solving followed by Instruction, PS-I, P!S-I-PS, CS1.

1. Introduction

Today, most education research agrees that active methodologies—whereby learners actively explore and construct knowledge—are helpful for learning (Prince, 2004; Freeman *et al.*, 2014). On the other hand, educators often have to teach specific introductory or technical concepts that students are unlikely to learn or even discover through free exploration. Informatics also faces this issue since it is a discipline with many technical aspects, especially in introductory programming (Guzdial, 2017). As a result, in introductory programming courses, a common approach remains directly teaching language elements, usually

☆This is an authors' pre-print version of the work. It is posted here for your personal use. Not for redistribution. The definitive version will be published in the Volume 20, Issue 4 (Special Issue on "The Role(s) of Abstraction in Computer Science Education"), December 2021 of the *Informatics in Education* journal. https://infedu.vu.lt/journal/INFEDU

*Corresponding author.

followed by their application in programming assignments. Direct instruction of technical concepts does not seem ideal for novice learners: it may bore them, also because they may not grasp the significance of the presented concepts from their perspective (Caspersen, 2018). This can result in low motivation and poor learning outcomes.

To tackle these challenges, in this paper, we propose a learning design specific to CS1. With CS1, we indicate "a first course in Informatics," in which usually students learn basic programming skills. To stimulate students' motivation and support their understanding, we suggest fostering "necessity-driven" learning. That is, challenging students with a problem that makes them "feel the necessity" of something they do not know yet.

Our pedagogical inspiration lies in approaches in which problem solving, as a preparatory activity, precedes instruction (PS-I) since this kind of approach can increase learners' motivation and improve understanding (Kapur, 2016; Loibl *et al.*, 2017). Furthermore, we draw on Productive Failure learning design (Kapur and Bielaczyc, 2012), which shows that failing in the preparatory problem-solving phase is even more effective for students to learn from the following instruction phase.

We will show an example of how our learning design can be used to support learning to program in CS1. In order to do this, we propose concrete examples of "necessity-driven" learning sequences and contextualise them within a CS1 course. Furthermore, since already in CS1 we recognise that abstraction—a fundamental idea[2] of Informatics—heavily comes into play, the examples are precisely set in learning moments when abstraction changes. These changes can be hard for novices (Curzon *et al.*, 2019, p. 533) since they require more effort and increase the cognitive load. Both upward and downward movements can be difficult for novices, and, since they determine different learning scenarios, the direction of the abstraction movement should be considered to develop effective learning activities with our learning design.

Section 2 reviews relevant literature focusing on CS1, abstraction in programming languages, Problem-based learning, Productive Failure, and PS-I approaches.

Section 3 presents a *learning design* for CS1 programming, which we call "Necessity learning design for CS1 programming" (NLD). Subsection 3.1 describes the "necessity mechanism", which is the core element of our learning design. Subsection 3.2 describes in detail our learning design, its features and its phases.

Section 4 proposes a concrete example of the application of our learning design in CS1 by using *necessity* sequences to support learning when abstraction changes (within the programming language of choice). Subsection 4.1 discusses how the choice of the learning path (i.e., which contents and their order) determines these movements and their direction (increase or decrease in abstraction). Subsection 4.2 shows how both directions pose challenges (albeit different) for learners and can lead to different teaching strategies. Subsection 4.3 proposes a CS1 learning path as an example to show where to place the *necessity* sequences we discuss. Subsection 4.4 shows four examples of learning sequences designed with NLD and presents the general structure of the examples as a tool for designing other *necessity* sequences.

Finally, section 5 discusses the relevance of our design and its future developments.

[2]For a historical and epistemological review and a discussion on the fundamental elements of an "informatical way of thinking", see Lodi and Martini (2021) and Lodi (2020).

2. Literature review

2.1. *CS1 and learning to program*

Students usually learn their first programming language in CS1 courses, together with some basic programming skills. For comprehensive reviews on introductory programming, see Robins (2019), Caspersen (2018) and references therein. In this section, we will draw mainly on those, highlighting some relevant considerations to our work.

Informatics education agrees that learning to program is difficult for many students. Moreover, there is a "lack of agreement among computing educators on the correct order in which to teach programming language concepts" (Robins, 2019, p. 351).

While often courses and textbooks mainly focus on—and are organised around—syntactical knowledge, it has been recognised that learning to program involves acquiring different types of knowledge (Bayman and Mayer, 1988; McGill and Volet, 1997): syntactic, conceptual, and strategic.

> **Syntactic knowledge** refers to the knowledge of the programming language syntax.
>
> **Conceptual knowledge** concerns program dynamics i.e., the knowledge of how programming constructs work and how this determines the execution of code, what is often referred to as notional machine [...].
>
> **Strategic knowledge** refers to the ability to apply programming syntactic and conceptual knowledge in order to solve new problems and achieve intended purposes. (Costantini *et al.*, 2020, p. 853, our emphasis)

As recognised by Robins (2019, p. 356), "the teaching of knowledge structures must be anchored in, and learning may most effectively emerge from, practical experience and examples." "This highlights again the need for well-designed example tasks and the practical opportunities for students to engage with them." Furthermore, the author recognises two essential features of programming tasks.

- Compiler feedback is "immediate, consistent, and (ideally) informative." We add that this can allow having impartial and objective feedback from the artefact one is creating.
- The "reinforcement and motivation derived from creating a working program can be very powerful."

Among the many theories and frameworks used or developed in Informatics education, we highlight elements from the *cognitive load theory* and discuss the *Learning Edge Momentum* hypothesis.

Cognitive load theory is a broad theory that describes how much a task loads a student's (working) memory. Often problem solving tasks can put a high load on students. Extensive research in Informatics education investigates the relationship between learning to program and cognitive load. Robins (2019) summarises four principles relevant in our study:

> The *worked-out-example effect* suggests that extraneous load is reduced by studying worked examples of problems rather than trying to solve the prob-

> lems from scratch, and similarly the *completion effect* suggests that load is reduced when the learner starts with partial solutions. Other examples include the *guidance-fading effect*, stating that novices need extensive support that can be reduced over time, and the *isolated/interacting elements effect*, stating that tasks with high element interactivity will be learned more successfully if elements are first introduced in isolation before being combined.
>
> (Robins, 2019, p. 344, emphasis as in original)

Moreover, according to Caspersen (2018, p. 117), "a good example must effectively communicate the concept(s) to be taught. There should be no doubt about what exactly is exemplified. [...] Conceptual knowledge is improved by best examples [...], where the best example represents an average, central, or prototypical form of a concept. To minimize cognitive load, an example should exemplify only one new concept (or very few) at a time."

Robins (2010) introduced the *Learning Edge Momentum* (LEM) hypothesis as an explanation for the fact that while there are many students who fail CS1, there are also many students who perform exceptionally well. According to Robins, "we learn 'at the edges' of what we already know by adding to existing knowledge. The more that new information is given a meaningful interpretation, the more effective learning appears to be" (Robins, 2019, p. 351). The hypothesis is that, given a certain target concept to be learned, acquisition of the earlier concepts facilitates (i.e., momentum towards success) acquisition of the later concepts. Similarly, failure to acquire the earlier concepts makes learning difficult (i.e., momentum towards failure). To use an economic metaphor instead of a physics one, one can think of compound interest, which quickly increases earnings because it is also calculated on previously earned interests. Robins (2010) argues that programming is a domain of highly integrated topics with clear and well-defined edges (like the pieces of a puzzle), and thus the effect will be very strong. According to the author, a positive momentum should be established from the beginning of CS1: "particular attention should be paid to the careful introduction of concepts and the systematic development of the connections between them" because "there is no point in expecting a student to acquire a new layer of complex concepts if the foundation of prerequisite concepts does not exist" (Robins, 2019, p. 360).

2.2. *Abstraction in Informatics and programming languages*

Informatics makes frequent reference to *abstract* entities and to activities identified as *abstractions*. The literature on Informatics epistemology (Colburn and Shute, 2007; Turner, 2021) and on programming languages (Gabbrielli and Martini, 2010), identifies *information hiding* as the main abstraction objective—layers of abstraction are built in such a way that layer n uses the functionalities of layer $n-1$ to provide functionalities to the layer $n+1$ via an interface which also hides the information needed to implement them. This helps manage the system complexity and allows for the independence and replaceability of a layer without influencing the others. This view applies to programming languages (e.g., from a high-level language like Java to its bytecode implementation, to assembly,

to the hardware machine), networks (e.g., the ISO/OSI stack), operating systems, and so forth. According to this view, a given programming language can be seen as a (specific) abstraction of the underlying physical machine.

It is important to observe that (raw) expressivity has little to do with abstraction. Machine languages and high-level languages have the same expressivity, being all Turing-complete, yet we assign to them different abstraction levels, based on the (intentional) hiding of information that happens when we move from a machine language to a high-level one (e.g., specific representations of numbers as bit sequences are hidden, or "abstracted away," when passing from machine to high level).

We should note, though, that it is possible to identify different levels of abstraction *inside a single programming language*. Each programming language provides *abstraction mechanisms*, which are "the principal instruments available to the designer and programmer for describing in an accurate, but also simple and suggestive, way the complexity of the problems to be solved" (Gabbrielli and Martini, 2010, p. 165). A `foreach` loop on a sequence is an abstraction from the `for` loop (of that same language), which would use explicit indexes on that sequence: the `foreach` hides those very indexes. In its turn, a `for` loop is an abstraction from a `while` loop because the `for` loop hides the explicit initialisation and increment of the index variable. In this paper, abstraction is therefore always connected to specific linguistic mechanisms.

Finally, observe that under this perspective:

- More expressive does not mean more abstract. We have already observed that expressivity and abstraction are in general orthogonal concepts. We remark here that even when expressivity is genuinely expanded, there are cases where there is no increase in abstraction. If we introduce first the *sequence* control structure, and then the *selection* control structure with the conditional command (`if-else`), we increase the expressive power of the language, but we do not have abstraction, because no information hiding occurs.
- More abstract does not always mean more general. In a C-like `for` construct we may specify almost arbitrary termination conditions and "increment" commands, and there is no constraint on the possibility of modifying the control variable in the body of the loop. Such a construct may be seen as the generalisation of Pascal-like `for` and `while`. However, it is *not* more abstract of them, because no information hiding occurs. Conversely, a `foreach` construct is more abstract and less general of its `while` "translation".

We can distinguish between two general classes of abstraction mechanisms in programming languages: *control* and *data* abstraction.

Control abstraction mechanisms "provides the programmer the ability to hide procedural data" (Gabbrielli and Martini, 2010, p. 165). Simple *control* abstraction comes into play early in a programming language learning path, for example, when dealing with structure control mechanisms like expressions, assignment, conditionals commands or iterative commands. Modern languages provide advanced mechanisms like procedures and exception handling constructs. For example, when dividing a program into subprograms (using functions or procedures), a programmer realises a functional abstraction, separating what the clients of such subprogram need to know to use it (e.g., name, parameters,

return type) and what they do not need to know (the body, i.e., the implementation of the function, which could be changed—for example, for efficiency reasons—without the client knowing it) (Gabbrielli and Martini, 2010).

Data abstraction mechanisms "allow the definition and use of sophisticated data types without referring to how such types will be implemented" (Gabbrielli and Martini, 2010, p. 165). Programming languages allow a variety of *data* abstraction mechanisms, hiding data representation details. These mechanisms range from simple ones, like the use of names to refer to memory locations, to the predefined language data types (collection of homogeneous, effectively presented values, with a set of operations on them), to more powerful mechanisms like the possibility to define *abstract data types*, up to all the data abstraction mechanisms provided by object-oriented programming.

From an educational perspective, the introduction of a new construct is often a movement across levels of abstractions. When, after a `foreach`, we unveil the possibility of an index-based iteration, we move down the abstractions. When we introduce functions, we provide a way to move up, and the same happens with the constructs for the definition of new data types. This movement across abstraction levels is a specific issue adding to the difficulty of learning. Students have to learn a new linguistic construct (or a new detail which was not previously introduced), its pragmatics when writing a program, *and* how this relates to the abstraction levels. Also choosing the correct construct (or the correct way to use a construct) is related to the abstraction levels. Students are instructed to prefer a `for` loop over a `while` for a sequential and complete scan of a sequence because the abstraction level of the `for` loop (the tool) matches the abstraction level of the "scan" (the problem). This "rollercoaster" over the abstraction levels is the example we tackle in this paper.

2.3. *Problem-based learning*

Problem-based learning (PBL) was first introduced in the 1960s at McMaster University. Traditionally, medical students had to memorise much information that they perceived to be superfluous to medical practice, but they were very involved when they actually worked with patients (Barrows, 1996). Therefore, they successfully experimented with a new methodology in which students are presented with problems (clinical cases) and have to study autonomously the material needed to understand and (possibly, but not necessarily) explain them. This methodology has been used in many other fields like Law, Social sciences, and Engineering[3].

According to a recent review (Bawamohiddin and Razali, 2017) on PBL for teaching programming, the problem is used to initiate and trigger the learning process. Problems must be ill-structured, real-world situated, complex, open-ended, motivational, unique, and solvable. Other researchers recognise similar characteristics (see, e.g., Oliveira *et al.*, 2013; Peng, 2010).

Drawing from medical education, PBL has been often implemented with the so-called "seven-step method" (adopted by many Informatics educators as well). The steps, as synthesised by Bawamohiddin and Razali (2017, p. 2036), are:

[3]For a recent comprehensive handbook, see Moallem *et al.* (2019).

1) terms and concept clarification;
2) problem identification;
3) brainstorming;
4) explanatory model sketching;
5) learning issue formulation;
6) self-learning and
7) information synthesising and testing.

Steps from 1 to 5 and 7 are conducted in small groups, while step 6 is individual study time. Usually, the process is quite long: for example, steps 1-5 can take from half an hour to several hours, step 6 (individual study) can take a week or more, and finally the last step can take other 2 to 10 hours (e.g., according to the proposals of PBL for programming, by Nuutila *et al.* (2008) and Bawamohiddin and Razali (2017)). Teachers act as facilitators: they guide students by motivating them, helping them understand the problem and guiding them in establishing the relevant learning objectives. On the other hand, they usually do not directly teach the target concepts.

PBL has been experimented with in Informatics courses, especially for programming. Reviews of experiences of using PBL show that it proved to be effective for introductory programming (see, e.g., Bawamohiddin and Razali, 2017; Oliveira *et al.*, 2013; O'Grady, 2012). In particular, Nuutila *et al.* (2008) successfully used the seven-step method in introductory programming courses. They found PBL very useful for replacing (at least partially) lectures. They respected the original PBL view from medical education: the focus is not on solving the problem but on autonomously set learning goals to acquire the knowledge needed to understand and explain "a case". Hence, they recognise that "some aspects of the programming skills require supplementary learning methods" like "supervised programming exercises [...] to teach the use of programming tools and effective work practices [... and also, at the end of the course,] a larger programming project" (Nuutila *et al.*, 2008, p. 64). However, they acknowledge that, because of the nature of Informatics and software development, elements traditionally more resembling "project-based learning", like focus on solving open-ended, multi-answer problems or working on complex, real-world tasks, have been integrated by researchers in PBL for programming. On the other hand, Kay *et al.* (2000) warns that calling the small and well-defined exercises used in conventional courses 'problems' (as many authors tend to do, as well as us) is not enough to claim to use PBL.

As we will discuss in subsection 2.4.2, our design is inspired by PS-I approaches such as *Productive Failure*, while having its own strong peculiarities (see 3.2.1). According to Falkner and Sheard (2019), "[w]hile associated in structure with problem-based learning approaches, productive failure has a specific emphasis on the use of failure as a pivotal point in the learning process". While sharing with PBL the essential feature of using an exercise for which students need to acquire more knowledge to motivate learning, our proposed learning design (described in Section 3) has significant differences (that will be clearer in the following). For example, our exercises are small and surgically designed around a specific target concept; there is no teacher scaffolding because failure is at the core of the learning design; the target concept is taught later with traditional instruction.

Deek *et al.* (1998, p. 314) discusses the so-called "alternative method" for CS1, that is "to introduce the problem in the lecture, engage the students in defining the statement of the problem, and allow the students to seek possible solutions independent of the programming language. Once the problem is solved, the language features necessary to implement the solution are presented. Finally, equipped with both the algorithmic solution, which the students develop, and the language syntax, the complete solution is translated into code and is then tested." This approach (which shows some, but not all, of the characteristics of PBL) is relevant to us because it shares with our design important features (e.g., a late direct instruction phase) of PS-I approaches (see the following 2.4.2). To it, we add the motivational aspect supported by the initial failure. Also, we do not explicitly distinguish between 'algorithmic design' and 'implementation' since the examples we propose are directly linked to language features. However, precisely because they are focused—as we will see—on the *necessity* of those constructs, we argue that our methodology can be helpful to stimulate *strategic knowledge* rather than just *syntactic knowledge* (see 3.2.2).

2.4. *Activities and difficulties that prepare for instruction*

In educational research, a growing body of literature argues that it is better not to start learning a concept from direct instruction.

For example, in *A Time For Telling*, Schwartz and Bransford (1998) describe a method in which undergraduate students analyse *contrasting cases* to develop prior knowledge that primes them to learn from direct instruction. Contrasting cases are presented side-by-side and consist of small sets of data, examples, or strategies. They recognise that "[n]oticing the distinctions between contrasting cases creates a 'time for telling'; [that is] learners are prepared to be told the significance of the distinctions they have discovered." Ultimately, they show that engaging in "analyzing contrasting cases [representing a target concept] provided students with the differentiated knowledge structures necessary to understand a subsequent explanation at a deep level [on that concept]."

In teaching statistics to advanced students, Schwartz and Martin (2004) provide further evidence to support their method (i.e., *Invention*) to prepare students before instruction. They demonstrated the efficacy of invention activities preceding direct instruction, despite such activities failing to produce canonical understandings and solutions during the invention phase.

Furthermore, Bjork and Bjork (2011, p. 57)—discussing the results of several psychology studies in which they took part between 1975 and 2009, together with what they learned from their teaching experience (also as teacher-researchers)—affirm the following.

> Conditions of learning that make performance improve rapidly often fail to support long-term retention and transfer, whereas conditions that create challenges and slow the rate of apparent learning often optimize long-term retention and transfer.

They use the expression *desirable difficulties* to describe those challenging conditions that are "desirable because they trigger encoding and retrieval processes that support learning,

comprehension, and remembering." Among these difficulties, Bjork and Bjork (2011) recognise the *generation effect*, that is "the long-term benefit of generating an answer, solution, or procedure versus being presented that answer, solution, or procedure".

About learners experiencing difficulties in preparatory activities, various forms of failure in activities preceding instruction are more and more investigated in educational research as a drive for better learning outcomes.

For example, VanLehn *et al.* (2003) conducted a study to help developers of intelligent tutoring systems understand which tutors' activities lead to success, confronting problem-solving episodes where tutoring does and does not result in learning a physics principle. They found that learning was infrequent when students did not reach an *impasse* (i.e., "when a student gets stuck, detects an error, or does an action correctly but expresses uncertainty about it") in problem-solving situations despite the tutor explicitly explaining the target concept. Conversely, "when students reach an impasse, they discover that they need to learn something, so they may adopt a learning orientation", making explanations more effective. Therefore, instructors should encourage impasses and delay instructional structure (e.g., feedback, questions, or explanations) until learners reach some form of failure and are consequently unable to proceed.

2.4.1. *Productive Failure*

Kapur and Bielaczyc (2012) leap forward in combining these two significant trends emerging from research—i.e., preparatory activities before instruction and failure as a drive to better prepare for learning—and propose the Productive Failure (PF) learning design. In the last ten years, Productive Failure has generated a considerable amount of research, much of which seems to confirm its effectiveness (Kapur, 2016; Sinha and Kapur, 2019).

PF integrates four interdependent mechanisms: "(a) activation and differentiation of prior knowledge in relation to the targeted concepts, (b) attention to critical conceptual features of the targeted concepts, (c) explanation and elaboration of these features, and d) organization and assembly of the critical conceptual features into the targeted concepts" (Kapur and Bielaczyc, 2012).

PF learning design develops in two phases. The generation and exploration phase—when students engage in complex problem solving and generate multiple *representations and solution methods* (RSMs), followed by the consolidation phase—when teachers organise and assemble relevant students' RSMs into canonical RSMs. Three core design principles guide PF in order to embody the cited four interdependent mechanisms.

1. Create problem-solving contexts that involve working on complex problems that challenge but do not frustrate, rely on prior mathematical resources, and admit multiple RSMs (mechanisms a and b);
2. Provide opportunities for explanation and elaboration (mechanisms b and c); and
3. Provide opportunities to compare and contrast the affordances and constraints of failed or suboptimal RSMs and the assembly of canonical RSMs (mechanisms b–d).

(Kapur and Bielaczyc, 2012, p. 49)

These core principles translate into many specific principles to guide the implementation of the two phases. We briefly report only those relevant to our work, all related to the problem-solving phase. "Designing the activity: 'sweet-spot' calibration of complex problems" involves challenging but not frustrating students. "Complexity of the problems" requires complex problems' scenarios allowing multiple RSMs. "Prior mathematical resources of students" states that the problem complexity also depends on students prior knowledge, around which problems must then be built.

2.4.2. *PS-I approaches*

In a broader perspective, Loibl *et al.* (2017) made a comprehensive attempt to summarise the features that define approaches in which preparing activities precede instruction, terming them PS-I. They consider Productive Failure and Invention as emblematic examples of PS-I approaches.

PS-I approaches involve an initial problem-solving phase in which learners are asked to develop solutions to a given problem. Then, the canonical solution and related target concepts are introduced in the following formal instruction phase. PS-I aims to most effectively combine these two core learning activities (i.e., problem solving and formal instruction) while preserving their strengths and limiting their disadvantages.

PS-I is different from other instructional methods with prior instruction because it demands learners to engage in problem solving before receiving the target knowledge. At the same time, the explicit instruction phase sets PS-I apart from other inductive methods—e.g., discovery learning (Loibl and Rummel, 2014) and PBL (see 2.3), where different forms of support guide learners to discover the target knowledge. In PS-I, by contrast, problem solving is not designed to acquire the target knowledge since that is what the instruction phase is dedicated to. The originality of this approach lies not in its constitutive elements, i.e., inductive problem solving and explicit instruction, but in combining them in a specific order.

In the problem-solving phase, students face a problem that requires applying the knowledge they have yet to learn. For example, in Glogger-Frey *et al.* (2015), practice teachers received learning diaries excerpts and faced the problem of inventing criteria to evaluate the use of learning strategies in them. The targeted evaluation criteria for learning diaries were introduced and discussed later during the following instruction phase.

Loibl *et al.* (2017) recognise two main variants of the problem-solving phase. One is that of Productive Failure approaches (Kapur and Bielaczyc, 2012; Loibl and Rummel, 2014), which require presenting data in a rich story that does not highlight the deep features of the topic and for which the solution cannot be intuitively guessed. The other variant is that of Invention approaches (Schwartz and Martin, 2004; Glogger-Frey *et al.*, 2015), in which contrasting cases serve to give students the relevant information.

They also distinguish two main variants in the implementation of the instruction phase. In Invention-like approaches, the canonical solution is presented without referring back to student solutions. In contrast, in PF-like approaches, the teacher starts from the students' solutions and uses them to explain the relevant elements of the canonical solution.

The efficacy of delayed instruction over instruction-first approaches has been shown across diverse learning domains and student populations by a substantial body of re-

search (Loibl *et al.*, 2017; Kapur, 2016). Such efficacy lies particularly in three cognitive mechanisms that need to be activated in learners during the problem-solving phase: prior knowledge activation, awareness of their knowledge gaps, and recognition of deep features of the problem (Loibl *et al.*, 2017).

In the last few years, given the consistent results supporting PS-I, research has investigated whether it is possible to increase this learning design's effectiveness further, examining different PS-I approaches. One of the leading research questions is whether the problem-solving phase should be scaffolded or not. According to Sinha and Kapur (2021), problem solving could be designed to nudge learners towards the canonical solution (i.e., success-driven scaffolding), towards sub-optimal solutions (i.e., failure-driven scaffolding), or to let learners experience failure without any form of scaffolding (in the problem-solving phase), resembling a straightforward Productive Failure design.

3. Necessity learning design

In the light of the reviewed literature, we propose a *learning design* to support the learning of programming in CS1, which we call "Necessity learning design for CS1 programming" (NLD). The core element of our design—which sits in the domain of PS-I approaches—is the *necessity mechanism*. What we call *necessity mechanism* is an original learning mechanism with various sources of inspiration (see 3.1) that shapes the learning experience to support motivation, engagement and understanding, by putting students in a situation that can stimulate in them the necessity of the concept that will be introduced afterwards.

In the following subsection, we analyse the *necessity mechanism*. Then, in subsection 3.2, we detail and discuss how we leverage this mechanism in our Necessity learning design for CS1 programming.

3.1. *Necessity mechanism*

The *necessity mechanism* consists of assigning students a carefully designed problem. The problem is built so that, from the one hand, students feel like they can solve it with the knowledge they already have. On the other hand, however, *necessity* problems are actually constructed to be unsolvable except with a particular concept (to which we refer to as the *target concept*, following PF and PS-I literature) not yet taught, making learners experience the need for it. When learners realise that, "surprisingly", they cannot solve the problem, it is the *time for telling*. That is when the feeling of strong necessity—generated by failing to solve the problem (or by struggling to find sub-optimal solutions)—can be leveraged to introduce the target concept.

For example, after students have learned the two main forms of definite iteration (i.e., sequential scanning (`foreach` loop) and loop with explicit but automatic index handling (`for` loop)) and after they have applied them to solve problems, a new problem requiring indefinite iteration is proposed. This new problem might be to count how many pseudo-random integers between 1 and 1000 a program generates before getting the number 42 (see necessity example 2). The problem is posed similarly to those faced when learning

definite iteration so as to inspire confidence in students that they can solve it with the loops they have learned so far. However, since they cannot meet the problem request—at least not easily nor optimally (e.g., they might use the `for` loop with a very large number of repetitions), students will hopefully feel a necessity (even an abstract one) of a construct that allows repeating until something occurs, rather than repeating a given number of times. The *desirable difficulty* of "having to resolve the interference among the different things under study [i.e., the interference between what worked for the previous known problems and what is not enough now] forces learners to notice similarities and differences among them, resulting in the encoding of higher-order representations, which then foster both retention and transfer" (Bjork and Bjork, 2011, p. 61) of the later instruction phase.

In designing *necessity* problems, we follow some of the PF principles on problem solving to "[c]reate problem-solving contexts that involve working on [...] problems that challenge but do not frustrate, [and] rely on prior [... students'] resources" (Kapur and Bielaczyc, 2012), in order to activate two of the cognitive mechanisms reported in subsection 2.4.2 (i.e., prior knowledge activation and attention to the target concept critical features).

To find the PF "sweet-spot calibration" of problems, we design them to be as similar as possible (i.e., using almost the same words, phrasings, scenarios, and requests) to the previous well-known problems students faced developing mastery of the previous target concept. Carrying on with the example, the last problem before the *necessity* problem (see example 2) might be to count how many times a program that generates 100 (or any fixed number of) pseudo-random integers between 1 and 1000 produces the number 42.

Most importantly, *necessity* problems are solved precisely by applying the new target concept, that is, the *smallest possible addition* to students' prior knowledge[4]. In other words—following the PF principle of relying on learners' prior resources—a *necessity* problem must be finely tuned to students' knowledge so that they can fully understand its request, identify its significant features, and devise strategies for solving it. However, none of these strategies will lead to the canonical solution as it requires the target concept to be developed. According to Bjork and Bjork (2011) on the *generation effect*, problems "can potentiate the effectiveness of subsequent study opportunities even under conditions that insure learners will be incorrect".

3.2. *Necessity learning design for CS1 programming*

In this section, starting from the *necessity mechanism* (which we envision usable more generally in science education), we propose a learning design specific to CS1, the Necessity learning design (NLD).

3.2.1. *P!S-I-PS necessity sequence*

Assuming the rationale of PS-I approaches (Loibl *et al.*, 2017), both to sustain motivation and because novices generally struggle to understand the significance of new concepts

[4]As already recogised by Shneiderman (1977, p. 195), "[a]t each step the new material [...] should be a minimal addition to previous knowledge, should be related to previous knowledge, should be immediately shown in relevant, meaningful examples and should be utilized in the student's next assignment."

when introduced by direct instruction—instead of planning to use the target concept "in the student's next assignment"[5], we create a "meaningful example" (i.e., the *necessity* problem), where the target concept is needed immediately *before* the instruction phase.

NLD is greatly inspired by unscaffolded PS-I approaches, which resemble straightforward PF (Sinha and Kapur, 2021), drawing on the idea that problem solving best prepares learners for the instruction phase.

On the other hand, NLD deviates from PF in how it introduces the target concept after problem solving, specifically by differing in the goal and implementation of the instruction phase. In the instruction phase of PF, the teacher introduces the target concept to the students (building on their RSMs[6]) and uses it to construct the canonical solution to the very same problem of the previous phase. By contrast, the instruction phase of NLD simply introduces the target concept and explains it in general without applying it to solve the problem, just showing its use in simple examples. After the instruction phase, NLD requires students to come back to the problem (whose goal was to make them experience the necessity of the target concept, see 3.1) that is precisely structured to be solved using the target concept just presented.

Inspired from the notation adopted by Sinha and Kapur (2019), we could describe our learning design as a *P!S-I-PS* approach. P!S emphasises that *necessity* problems are unsolvable (!S) before the instruction phase. The trailing PS indicates a second problem-solving phase, in which students return to the same problem after the instruction phase.

P!S. The problem posed in the P!S phase diverges from the PF in the "complexity of the problem" design principle. PF designs problems to be complex and information-rich (providing even unnecessary data) so that it is natural and inevitable for students to generate multiple RSMs. That is because of the key role of students' RSMs in the "consolidation phase", in which PF builds *direct instruction* on "organizing and assembling the relevant student-generated RSMs into canonical RSMs" (Kapur and Bielaczyc, 2012). While programming problems also admit various RSMs (always sub-optimal without the target concept, as discussed later in 3.2.3), the Necessity learning design does not rely on the students' RSMs to introduce the target concept, nor does it build the canonical solution together with students in the instruction phase.

Among the reasons for this different sequence is the nature of problems faced in learning programming. The solution to a programming problem is a program, and a program is an interactive object. The student who develops a program can immediately check whether or not her solution can be executed. A non-executable program is a first obvious indication of an error. On the other hand, if a program is executable, there are often many possibilities for the student to verify whether it works correctly. In other words, the student's program is an interactive solution attempt, which returns information helpful for solving the problem. This unique characteristic of programming problems constitutes a source of motivation for students to experience first-hand the use of the target concept to solve the *necessity* problem.

[5] See again the quote in footnote 4.

[6] Representations and Solution Methods, see 2.4.1.

Furthermore, a specific consideration can be made for CS1 programming. Since it involves introductory and mostly technical knowledge, there is inevitably less room for various RSMs and for debating on them. On the one hand, confronting and discussing different solution strategies (before instruction but also after the second PS phase) is undoubtedly valuable. However, on the other hand, the actual learning trigger of our learning design is not the discussion about students' RSMs but the feeling of necessity introduced by the *necessity* problem.

While it would be unnatural (because of the introductory and mostly technical knowledge) to design CS1 complex problems with multiple RSMs, we focus on developing meaningful but minimal problems that are "surgically" precise with respect to the target concept. *Necessity* problems for CS1 programming are built on students' prior knowledge but in such a way as to be solved only with the target concept, as if the problem were an encrypted message and the target concept the key.

I. As anticipated, in the instruction phase of PF (and, more generally, of PS-I) learning design, the teacher illustrates the target concept and then applies it to solve the problem. In our scenario, providing the solution would waste a precious learning potential, that is, the feeling of necessity (generated by the *necessity mechanism*, see 3.1) students have experienced in the P!S phase. The optimal strategy formulated by VanLehn *et al.* (2003), whereby instructors, after students reached an *impasse* (*P!S* phase), "prompt them to find the right step [...], and [...] provide an explanation only if they have tried and failed to provide their own", supports the choice of not revealing the solution already in the instruction phase (if we equate 'providing an explanation' with 'writing a program').

PS. In addition, all the reasoning students did in trying to solve the problem before knowing the target concept can be valuable not only to understand the concept's significance but also to realise how to apply it to solve the problem. Indeed, applying a concept is a further and more challenging step than simply understanding it. From a learning programming perspective, understanding a concept corresponds mainly to the conceptual level, knowing how to apply it to the strategic level.

See Table 1 for a summary comparison of NLD with Productive Failure.

3.2.2. *Learning between necessity sequences*

Necessity learning design (NLD) does not aim to structure all phases of a CS1 programming learning path. In other words, the sequence (i.e., P!S-I-PS) of *necessity* activities is not sufficient in itself for students to fully develop the syntactic, conceptual and strategic knowledge of the related target concept. For this to happen, it is essential that, after every occurrence of a *necessity* sequence, students are exposed to significant examples of the related target concept and other problem-solving situations[7] and have enough time to develop mastery. More generally, educators should adopt all the best practices that Informatics education research suggests to fully develop the three levels (i.e., syntactic, conceptual and strategic) of knowledge.

[7]Recall that the problems students tackle in this phase are similar to the *necessity* problem of the next sequence, see 3.1.

Table 1
Comparison between Productive Failure (unscaffolded PS-I) and Necessity learning design (P!S-I-PS)

Productive Failure (PF)	***Necessity learning design (NLD)***
PS	**P!S**
Problem built on students' prior knowledge	
Problem sweet-spot calibration: challenging but not frustrating	
Problem: complex, information-rich and ill structured (what-if scenarios)	Problem: simple, well structured, very similar to those the students are already used to
Problem stimulates multiple RSMs	Problem surgically designed to stimulate the necessity of the target concept
No teacher scaffolding	
Students discuss to confront different RSMs	RSMs are programs, tinkerable objects (self assessment and trial & error)
Students present their work and compare (guided by teachers) affordances and constraints of failed or suboptimal RSMs to assemble the canonical RSM	–
I	
Teachers introduce the target concept in the context of the problem and solve it	Teachers introduce the target concept without referring to the problem (showing the solution would waste the "necessity learning potential")
Students practice on well-structured problems on the target concept	–
Teachers formally introduce the concept and explain it through simple examples	
Students practice on isomorphic problems	–
–	**PS**
–	Students immediately apply a concept (strategic knwl) after learning it (conceptual knwl) in a meaningful and well-known context (P!S problem)

The characteristic of NLD to focus on crucial learning moments does not undermine its more general scope nor its usefulness. Indeed, precisely because NLD immediately stimulates the use of the target concept in a problem-solving context—in which it is "the right thing to use"—NLD aims to be an ideal starting point for developing especially the strategic knowledge for that concept.

3.2.3. *Hard vs. Soft necessity*

The *necessity* of a programming concept (and the related construct) can be considered from two different perspectives.

From the programming languages perspective, there is only one truly *hard* necessity—that of the brute-force ability to program a specific function. It shows up only when the (subset of the) language in use is not Turing complete. If we allow only (true) definite iteration, then, for example, we cannot write programs that may loop forever for certain input values. However, modern languages are all Turing complete (at least in their "standard model" (Martini, 2020) where arbitrary resources are allowed). After all, besides the ability to memorise data of potentially unbounded size, only selection and indefinite iteration

(or recursion) are needed for Turing completeness.

On the other hand, Turing completeness does not tell all the story. The brute-force ability to program any computable function usually passes through an unnatural coding of data and processes. Lists, for instance, are not needed for such completeness. We may always encode a generic list of integers $[x_0, x_1, x_2]$ with *a single* integer $2^{x_0} * 3^{x_1} * 5^{x_2}$ (*Gödelization*) and implement all list operations through prime factor decomposition. However, this possibility is irrelevant in a CS1 programming learning context.

Therefore, we better consider an educational perspective and identify an "educational" *hardness* of *necessity*, which shows up exactly when the available programming tools express the new concept (i.e., the target concept) only through unnatural or too complex coding. As a matter of fact, it is unlikely for CS1 students to be able to produce such advanced—yet unnatural and complex—solutions at the point where they are in the programming learning path. This educational perspective is bound to the chosen programming language and the order of topics in the learning pathway.

Therefore, an *educationally hard necessity* of a target concept appears when students, drawing only on what has been taught in the course so far—hence without the target concept, can produce only sub-optimal solutions to a given problem. Most of the time, using in a "standard way"[8] the constructs learned until that moment, students will also be able to produce partial solutions, i.e., solutions that do not solve the problem in all possible cases. However, recall that—apart from the exceptional cases of a Turing-incomplete language (subsets)—complete solutions (i.e., solutions that work for all cases) are always technically available, although through more or less fancy hacks. Nevertheless, as said, it is very unlikely (though not impossible) that a student could circumvent an *educationally hard* necessity problem. For example, students might "force" a definite loop construct they know to express an indefinite iteration, e.g., in Python, using a `for` loop over a list and increasing the list length in the repetition body[9]. Solutions like this would be a sign of great mastery of the concepts preceding that moment of *necessity*. In these (rare) cases, teachers should make these students realise that such solutions, though workable and clever, are nonetheless sub-optimal hacks (often inelegant, inefficient, unnecessarily complicated) and prone to create issues as the complexity of tasks increases. Students who are able to circumvent a *hard necessity* problem are also likely to be receptive to such clarifications.

By contrast, in cases of *educationally soft necessity*, sub-optimal solutions formulated without the target concept can, quite easily, solve the problem completely relying on students' previous standard knowledge. In such cases, it is not unlikely that students will be able to circumvent a *soft necessity* problem since they could "blindly" use what they already know. For example, with reference to the turtle geometry sequence (see example 1), drawing a polygon of 20 sides can be done by blindly replicating the same code 20 times. In general, in *soft necessity* cases, sub-optimal solutions may be fully functional. However, they are always one or more of the following: less compact (e.g., because of repeated

[8] The canonical usages of a construct, as taught to students.

[9] Typically, in most modern languages, constructs designed for definite iteration (e.g., `for`, `foreach`) can be forced to express indefinite iteration, sometimes in more natural ways (e.g., in C), sometimes in unorthodox and inelegant ways (such as the Python example just mentioned).

code)—and therefore less maintainable and more prone to errors, less modular (e.g., not using functions, not adopting OOP), less clear (e.g., intricate solutions), and also possibly less efficient (i.e., more computationally costly in time, space or both).

From the educators' perspective, being aware of whether the *necessity* is *educationally hard* or *soft* helps design the related *necessity* problem. If it is *soft*, the *necessity* problem should be structured in such a way that circumventing it by blindly using prior knowledge is as difficult as possible. Conversely, if it is *hard*, the *necessity* problem should require exactly what is (educationally) impossible to do without the target concept. To sum up, the goal is always to stimulate the necessity of the target concept, without which it must be (almost) impossible (i.e., *educationally hard necessity*) or hindered (i.e., *educationally soft necessity*) to devise a complete solution.

To clarify in a concrete context, we refer to a scenario where the problem is to input a list of students' names and randomly choose the first to take the oral exam.

1. If the number of students is known...
   a) ... and it is very small.
      No *necessity* is stimulated since it should be fairly simple for CS1 learners to use separate variables to input the names, extract a random number, and use a simple selection (with few `elseif` branches) to print the corresponding student.
   b) ... and it is long.
      It is still conceptually easy to solve the problem with unrelated variables and a (long) selection with many `elseif` branches. However, the solution code is long, repetitive, and errors are likely to be made. Even though it is possible to solve the problem completely, writing such a solution is prone to errors and could be frustrating. This scenario can stimulate in students a *soft necessity* of some type of collection with index access (like arrays).
2. If the number of students is unknown.
   An *educationally hard necessity* is stimulated. Students can produce an *incomplete* solution assuming a large maximum number of students, leading back to the solution of the previous point. Of course, this solution would be highly suboptimal. In this case, we talk about an *educationally hard necessity* (this time, of some type of dynamically extendable collection with index access, like Python lists) because students have no "standard"[10] way to produce a *complete solution*, i.e., to memorise an arbitrary, unknown number of students with only simple variables.

To summarise, the *hardness* (and the effectiveness) of any *necessity* sequence is determined by the chosen language and the specific order of the topics in the learning path. For this reason, information in each of the presented *necessity* examples always includes "What students already know" and, when relevant, a warning on other concepts students should not know yet.

[10] However, see the above discussion on how to simulate a list with a single integer variable.

4. A use of NDL in the CS1 abstraction rollercoaster

In the previous section, we presented and described NLD. In this section, we present a possible application of NLD in some moments that we consider crucial in a CS1 course, that is, those in which the level of abstraction changes in relation to the constructs of the chosen language (see 2.2).

After discussing these movements of abstraction (4.1) and the relevance of their direction (4.2), we present a possible learning path for CS1 (4.3). Finally, we show four *necessity* examples (4.4) within that learning path, also detailing the general structure of the examples.

4.1. *Abstraction movements in CS1 programming*

The history of programming languages, from low level to high level, clearly shows an abstraction process, with the introduction of more abstract constructs (for the notion of *type*, see Martini (2016a,b)), or with the creation of more abstract languages, sometimes maintaining low-level functionalities (even when not strictly necessary for expressiveness), other times sacrificing them for cleanness.

The creation of these abstractions in programming languages is a complex process, driven both by experiment and semantics. In Visser (2015), we find a discussion about this abstraction creation process. First, a *programming pattern*—a "recipe" for solving a re-occurring problem, which the programmer applies manually in any instance (e.g., calling and returning sequences in assembly language using the return stack)—is identified. Then, a linguistic *abstraction*—a construct providing a "black-box" for that pattern (e.g., functions and their parameter passing mechanisms)—is devised and created. The essential point is that a "good" abstraction, once created, gets autonomous life because it captures an important concept of software development.

Professionals have welcomed this movement upward the abstraction ladder—because it enables them to express more and more complex computations in a simple, evocative, concise way—and resort to lower-level constructs (or languages) only when necessary (e.g., for efficiency's sake).

By taking an expert perspective, it could be tempting to *always* take the same direction (usually called a *bottom-up approach* (Caspersen, 2018, p. 113)) and follow an ascendant abstraction path when teaching programming to students. That is, starting from lower-level constructs so that, when a higher-level construct (abstracting the former ones) is introduced, students can fully understand its underlying details. However, as already noted by Shneiderman (1977)—and in line with the studies on cognitive load theory (see 2.1)—students "would be overwhelmed if the general form were presented first, [while they] can absorb complex forms in a step by step process", therefore "[n]on-essentials must be stripped away so as to provide students with a minimum useful subset of the language which can be expanded gradually." Indeed, a small subset of abstract constructs (e.g., `print`, `if`, `foreach` loop) is enough to produce early functioning and meaningful programs (e.g., given a sequence of strings representing the XML code of a social network

posts, print out only those mentioning your name), and thus sustain motivation in novice learners (Caspersen, 2018, p. 113). Furthermore, some studies show that implicit looping is more natural for novices than explicit looping (Guzdial, 2008), hinting at the possibility to teach more abstract constructs before the less abstract ones (i.e., focusing more on "what" than on "how", following Statter and Armoni (2020)).

Hence, in some cases, it is possibile for educational purposes to take the abstraction ladder downward, in the opposite direction of the one followed to introduce more abstracts constructs in programming languages. For example, instead of going upward from using the `while` loop for *definite* iterations (i.e., by manually handling an index/counter), to the `for` loop with explicit but automatically handled index, up to the `foreach` loop with implicit (and hidden) index handling, we can take the opposite route by starting to teach the more abstract loop (i.e., the `foreach` loop) and going down from there. Of course, CS1 students will soon face situations in which the most abstract construct is not sufficient anymore (or it is less elegant, simple, efficient). When this happens, students can feel the *necessity* (see 3.1) of "opening the black box" to have a less abstract but more powerful mechanism.

On the other hand, during a CS1 course, adding more constructs often raises the abstraction level. For example, in our proposal this happens when introducing a dictionary-like data type (i.e., a mapping between two finite collections of arbitrary types). Since it helps solve problems handled before with parallel arrays (one containing the indexes and one the values), introducing dictionaries can be seen as an increase (compared to using parallel arrays) in data abstraction.

We observe that the order in which concepts are introduced determines the direction of the movement of abstraction. In the previous example on loops (i.e., moving down towards the less abstract `while` loop), the abstraction movement would be upward if the general form (i.e., `while`) were presented before the more abstract one (i.e., `foreach`).

As mentioned in 2.1, there is no agreement in the literature on what is the best topic order to follow in CS1. Consequently, we observe that CS1 courses go both up and down in abstraction. Also, introducing the same concepts in different courses may correspond to opposite directions in the abstraction movement due to different choices on the topic order.

Hence, considering the multiple changes of abstraction level across the abstraction mechanisms of any programming language for CS1, we recognise what we call a *rollercoaster of abstraction*.

In 4.3, we present a complete CS1 learning path and discuss the abstraction movements (and their directions) within it and also the possibility of alternative paths (and thus different abstraction movements and directions).

4.2. *Abstraction ups and downs: different and difficult*

As anticipated at the end of 2.2, movements between levels of abstraction pose specific learning challenges.

Moving downward the abstraction ladder is not easy because novices have a hard time dealing with many details (see 2.1 on cognitive load). A drop in the level of abstraction

requires students to consider additional information, determines a less simple way to do things but allows for more sophisticated computations. A metaphor that speaks well of this difficulty is the car transmission. It is intuitive to understand how a person who has learned to drive automatic cars finds it difficult to drive a manual car because it requires knowing and manoeuvring more details.

However, moving upward the abstraction ladder seems, more surprisingly, not easy either. A study conducted by Alexandron *et al.* (2012, p. 157) on *live sequence charts*

> showed that some [students] felt that the high abstraction level does not give them enough control, though the goals of their program were achieved without getting into lower level details. [... T]his subjective feeling is strongly related to [...] students' previous programming experience. Since the students were used to working on lower abstraction levels, it determined their perception of what the 'right abstraction level' is. When moving to a higher abstraction level, they could not control things that they used to control before, and thus felt that they lose power.

It is not uncommon for people who are used to driving manual cars to report difficulties (at least at first) in switching to an automatic transmission, complaining of less control (it is not possible to control the engine brake in the same way as in a manual car) and a feeling of disorientation.

Therefore, since riding the abstraction rollercoaster of learning CS1 programming seems difficult both upward and downward, we propose examples of how to use NLD to support learning at these critical abstraction movements. We think NLD can help in these critical learning steps, also thanks to its multiple influences (discussed in 2—e.g., the constructivist idea of "generation effect" for solid and deeper learning and to favour the learning edge momentum; Productive Failure to best prepare for instruction).

Moreover, recognition of the direction of the abstraction movement helps guide the design of the *necessity* sequence. Indeed, whether abstraction goes up or down, the nature of the *necessity* problems changes since scenarios of different *necessities* arise.

Reducing the level of abstraction causes difficulties for students because it requires them to see and deal with previously hidden details. We observe that in these cases, it is useful (and generally straightforward) to stimulate a *hard necessity* (see 3.2.3) in them, that is, to confront them with a problem that is (educationally) impossible to solve without those additional details.

In a different way, increasing abstraction also causes difficulties for students since it takes away details they learned to control. In these cases, it is difficult to find hard necessity scenarios: usually, more abstract tools are intended to facilitate the use of more trivial tools rather than to enable new functionalities (see 2.2). Consequently, when abstraction increases, it is appropriate to stimulate a *soft necessity* (see 3.2.3) to use the more abstract tool by constructing problems that make it difficult (i.e., time-consuming, tiring, prone to errors) using "blindly" the less abstract tools already possessed by students, thus showing the opportunity to resort to the more abstract one.

We consider more straightforward the development of problems supporting learning when the abstraction goes down. Therefore we provide only one example of the NLD use

at a downward movement (example 2) and three examples of NLD supporting an upward movement (examples 1,3,4). All examples are framed into a CS1 learning path, which we describe in the following section.

4.3. *A possibile CS1 learning path*

In the following, we report the contents of a CS1 learning path in which the examples provided in subsection 4.4 are placed.

The proposed course follows a rather classical approach to CS1, based on the CS1 for Math major that we have successfully experimented with over the last ten years, both with traditional in presence lectures and as a synchronous online course (Lodi *et al.*, 2021).

Compared to an entirely classical progression (as proposed in some textbooks, e.g., Guttag (2021), Downey (2015)), note the `turtle` module in the first lessons, used as a tool for a playful and creative introduction to programming (influenced by Papert's constructionist approach with Logo turtle geometry (Papert, 1980), of which the `turtle` module is a Python implementation). Indeed, as in renowned courses (e.g., Harvard's CS50[11] or Berkeley's CS10[12]), this introductory part can be approached with visual languages such as Scratch or Snap!, which provide turtle primitives and simple repetition constructs such as `repeat N`.

Here our proposal of a CS1 pathway follows, instantiated with Python language but easily adaptable to any other high-level imperative language. The path includes main programming concepts and constructs, together with Python language features realising them, as well as elementary patterns, notable algorithms, and theoretical aspects. Along the path, the target concepts of the *necessity* examples provided in 4.4 are indicated in bold italics.

Example of CS1 learning path for non-majors

- Importing constants and functions from a module
- Basics of the `turtle` module
  - `forward` (`backward`) function
  - `left` (`right`) function
- `for` loop to express a simple `repeat N` with `N` being a fixed integer value (e.g., `for i in range(10)`) – ***example 1 target concept***
- Built-in data types
  - Integers, floats
  - Strings and their basic operations (including selection of a character)
- Variables and assignment
- Functions (calling them, passing paramenters, defining custom functions)
- Input and output (`input` and `print` functions)
- Type conversions (e.g., `int()`, `str()`)
- `random.randint` to generate a random integer in an interval

continued on next page

[11] https://cs50.harvard.edu/college/2021/fall/syllabus/#lectures
[12] https://cs10.org/su21/5syllabus/#welcome

- Boolean type and boolean expressions
- Conditional commands (`if`, `if...else`, `if...elif...else`)
- `for` loop to iterate over sequence elements ("`foreach`" loop)
- Elementary pattern: linear scan (for each element of a sequence, do some operations)
- Tuples (and generalised assignment)
- Slices on sequences (`[start:stop:step]`)
- Elementary pattern: linear scan with a gatherer
- Idea of object and methods
  - `is` vs. ==
  - Methods on built-in types (e.g., `str`, `float`)
- `range` and `for` over a `range` to express the definite iteration with explicit but automatically handled index (e.g. `for i in range(1,10,2)`)
- Elementary pattern: linear search
- `while` loop to express indefinite iteration – ***example 2 target concept***
- Algorithm: Euclidean algorithm to compute GCD
- Pattern/algorithm: binary search
- Computational complexity: linear vs. logarithmic
- `matplotlib.pyplot.bar` to plot a simple bar chart
- Arrays (or lists) with index access – ***example 3 target concept***
- Lists (mutability, dynamic insertion and deletion of elements, list methods)
- Recursion
- Computational complexity
  - resources, computational steps, cost of elementary/non-elementary steps, big-Os
  - complexity of binary search
  - complexity of numeric algorithms
- Sorting algorithms: insertion sort, quicksort, merge sort
  - their computational complexity
- Dictionaries – ***example 4 target concept***
- List comprehension and dictionary comprehension
- Generators
- Object-oriented programming (OOP)
  - Classes and objects, attributes and methods
  - "Magic" methods
  - Private attributes, instance attributes
  - Subclassing and inheritance
- Binary tree abstract data type (implemented with OOP)
- The Halting Problem

This pathway is only a proposal, and we provided it to place the *necessity* examples in a concrete context. Indeed, the examples we developed with NLD can fit into other CS1

learning paths, provided that the prerequisites are met and the topics students should not know (to stimulate the various *necessities* across examples) have not been addressed.

With the same warnings about dependencies between topics, it is important to note that other topics can be addressed with NLD. In fact, these presented here are only a few examples of the many possible *necessity* sequences. For example, educators might use NLD to stimulate the necessity of a coherent representation of an object instead of relying on unrelated data structures and functions to introduce object-oriented programming. This transition corresponds to a movement of (data and control) abstraction upward, which can be set in a *soft necessity* scenario.

4.3.1. *Ups and downs in our and in other paths*

From the viewpoint of abstraction movements, our learning path immediately shows some ups and downs. In the beginning, we intuitively introduce the `repeat N` loop (with `N` being a fixed integer known at compile time). The `repeat N` loop is realised in Python with a `for i in range(N)`. Leaving initially unexplained what `i` and `range` are, we teach a simplified use of the construct that, in this case, serves only to repexat `N` times a block of instructions. At this point the abstraction goes up a bit, because the first "real" Python loop we teach after the `repeat N` loop is the `for` on sequences (`foreach`). After that, the abstraction goes down, as we explain what a `range` is and how to use the `for` construct to iterate over a `range`, to have (in Python, to simulate) a `for` loop with explicit but automatically handled numeric index. The goal is to access sequence elements by index (and not just *repeat N times* a block of instructions). A *hard necessity* sequence can be constructed around the need to use indexes in a non-trivial way, e.g., to access the previous/next element or only certain parametric positions. Abstraction goes down further when the `while` loop is introduced to express indefinite iterations (see example 2).

The rollercoaster described so far is experienced by students not only at the crucial moment of learning those new constructs/concepts. Indeed, students, in order to solve future problems, will always have to move between the different levels of abstraction encountered to choose the construct at the most suitable level for the problem faced. Having made this necessary clarification, we observe how abstraction tends to increase or remain constant from this point onwards in the proposed learning path.

However, since the path in 4.3 is only a proposal and no correct topic order is established, it is easy to imagine other learning paths with other downward moments, showing that the rollercoaster is inherent in CS1 programming.

For example, Python provides a powerful tool for "inline" list construction, *list comprehension*[13]. This construct is more abstract than the classic elementary pattern that uses a `for` loop and an empty list as a gatherer. In addition, it features a syntax inspired by the mathematical notation of set construction (i.e., note the similarity of $\{x^3 \mid x \in [0..4]\}$ with `[x**3 for x in range(5)]`). It has been found that constructions such as these are intuitively much preferred by non-programmers (Pane *et al.*, 2001, p. 258). It

[13] According to Python documentation, list comprehensions "provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition."

is possible to consider introducing this list-building construct first, a choice that might make particular sense for Math-major given the similarity of list comprehension with the intensional mathematical representation of sets. The necessity of learning the `for` loop, thus going down in abstraction, can then be stimulated when creating the list is not the only thing that needs to be done, but other operations also need to be performed at each iteration.

Generally, an approach that starts from more abstract constructs—thus deviating from the more established tradition of teaching programming from the basics and abstracting from there—might be beneficial in domain-specific contexts other than Informatics. If researchers or professionals outside Informatics need to learn to program, likely, the level of abstraction of the tools they are interested in is the one closest to the type of data they have and the elaborations they need to do.

According to this perspective, for instance, it would be possible to learn the *tree abstract data type* first and only afterwards, when the necessity of modelling more sophisticated (than trees) data structures, learn how trees can be implemented through lists. Similarly, as with list comprehension, it is possible to teach a powerful and useful construct such as *generators* first in their most abstract and intentional form provided by Python. Afterwards, it will be possible to descend the abstraction ladder to teach generators' explicit construction and the more sophisticated commands (e.g., **`yield`**) when necessary (e.g., when a mathematician needs to handle a large list of primes without allocating it all in memory).

4.4. *Examples of NLD use in abstraction movements*

In the following subsections, we present four examples of the use of Necessity learning design. The proposed examples are designed for Python, but—unless specific limitations are reported—they can be adapted to introduce the same concepts in other imperative languages as well.

The examples presented below are *necessity* sequences. A *necessity* sequence is the succession of three phases: P!S, I and PS. Each sequence, designed according to NLD, aims to teach a concept that determines a change in the abstraction level in the use of the programming language.

Each example is presented in a box divided into three parts: *necessity* scenario, *necessity* sequence, and example's characteristics.

The *necessity* scenario provides the following information.

- **Title**. It evokes the necessity stimulated by the sequence.
- **Problem in a nutshell**. A concise description of the problem posed to stimulate the necessity for the target concept.
- **What students already know**. It lists the knowledge students should have to deal with the sequence, expressed both at the syntactic and conceptual level and the strategic level (see 2.1).
- **Target concept**. The new minimal addition to previous knowledge we want the students to learn expressed both on the syntactic and conceptual level and the strategic one (see 2.1).

After the first dashed line, the second part describes the *necessity* sequence in detail.

- **Before the necessity sequence**. It reports an example of a task to which students have become familiar in developing mastery of the prior knowledge. This task is the last before the P!S phase and is structurally and linguistically very similar to the *necessity* problem of the current sequence (see why in 3.1).
- **P!S – Problem-solving phase (unsolvable problem)**.
  - *Problem*. The text (as proposed to students) of the *necessity* problem (see 3.2), a problem that requires the target concept in order to be solved. It may consist of a sequence of tasks.
  - *Necessity trigger(s)*. The obstacles students encounter when trying to solve the problem without the target concept. Not being able to overcome these obstacles should stimulate the feeling of need for the target concept.
  - ***Necessity***. The core mechanism of our learning design describes the feeling of the necessity of the target concept. That is, what students experience by not being able to solve the proposed problem.
  - *Sub-optimal solution(s)*. The solutions that students might develop trying to solve the problem without the target concept. A solution may be incomplete or complete (i.e., a solution that works for all cases but is still sub-optimal for other reasons, see 3.2.3). The solutions reported do not necessarily have to emerge, nor is it required that all students are able to formulate them.
- **I – Instruction phase**. The target concept, and its related knowledge, that the teacher directly instructs to students (without applying it to solve the necessity problem).
- **PS – Second problem-solving phase**. It shows the optimal target code students should be able to develop engaging again in the necessity problem after being instructed on the target concept.

After the second dashed line, the third and last part reports some characteristics of the example and, when present, implementation's warnings.

- **Characteristics**.
  - *Abstraction mechanism*. It indicates whether the sequence is about *control* abstraction or *data* abstraction (see 2.2).
  - *Abstraction movement*. It indicates whether the target concept represents an *increase* or *decrease* in the abstraction level (compared to what students can do with the tools they already know).
  - *Educational hardness*. It indicates whether an educationally *hard* or *soft* necessity is stimulated (it depends on whether or not it is possible to develop complete solutions without the target concept, see 3.2.3)
- **Warning(s)** (optional). It lists, if any, the knowledge that, if possessed by students, would allow them to easily solve the problem, not letting them experience the need for the target concept.

Each example is followed by a short discussion where we report some important considerations on the related necessity sequence.

4.4.1. *The necessity of definite iteration*

Necessity example 1

The necessity of definite iteration (with turtle geometry)

Problem in a nutshell

Drawing a polygon with a given (high) number of sides. Then, changing the program to modify the side length. Finally, changing the program to modify the number of sides.

What students already know

- *Syntactic level + viable conceptual model*
  - Importing functions from libraries
  - Basics of the `turtle` module:
    - `forward` (`backward`) function
    - `left` (`right`) function
- *What students already know how to do (strategic level)*
  - Using `forward` (`backward`), `left` (`right`) functions from `turtle` module to draw contiguous segments

Target concept

- `repeat N` loop, with `N` being a fixed integer known at compile time (syntactic + conceptual)
- repeat instructions a fixed, pre-determined number of times (strategic)

Before the necessity sequence

Example of a previous task not needing the target concept

Draw a square of side 50.

P!S – Problem-solving phase (unsolvable problem)

Problem

The problem consists of three sequential tasks. The next task is given when students have finished the current one.

A. Draw a 20-side regular polygon of side 50.
B. Edit the previous program so that it now draws a polygon (still 20-side) with sides length equal to 45.
C. Edit the previous program so that it now draws a 18-side polygon (with sides length equal to 45).

continued on next page

Necessity triggers

A. Students need to repeat 20 times a two-instruction block.
```
forward(50)
left(18)
```
B. Students need to update 20 times the side length.
```
forward(45)
left(18)
```
C. Students need to remove two blocks and update 18 times the angle.
```
forward(45)
left(20)
```

Necessity

I need a way to repeat a block of code a fixed number of times.

Sub-optimal solution

- (complete) The straightforward solutions with the replicated code are completely correct, but tiring and likely subject to errors or oversights.

I – Instruction phase

Illustrating how to *repeat N times* a block of instructions, with N being a fixed integer value, realised in Python.

```
for i in range(N):
    <block of code>
```

PS – Second problem-solving phase

Target code

A possibile solution to task C (solutions to taks A and B can now be easily obtained by modifying this code as well).

```
from turtle import forward, left
for i in range(18):
    forward(45)
    left(20)
```

continued on next page

Characteristics

- *Control* abstraction
- *Upward* abstraction movement (compared to replicating the same block of instructions many times)
- *Soft* necessity (complete, though sub-optimal, solutions are possible without the target concept)

Implementation warning

Other concepts that students should not know yet for the mechanism to work:

- Variables and assignments

Example discussion

Task A would be sufficient to introduce the necessity of definite iteration. However, we believe that tasks B and C can reinforce this necessity. Indeed, the request for multiple changes aims to provoke a little frustration in students so that they can feel the need for a repetition construct, especially in a case like this of soft necessity (see 3.2.3).

Note that this would not be the case if variables and assignments were already known to students. In that scenario, tasks B and C are not useful. Therefore, task A should be designed in such a way as to stimulate the necessity of definite iteration even more (e.g., asking for more polygons with more sides) since using two variables (i.e., one for the side length and one for the number of sides) would make the updates required by tasks B and C immediate.

4.4.2. *The necessity of indefinite iteration*

Necessity example 2

The necessity of indefinite iteration

Problem in a nutshell

Counting how many random numbers are generated before getting a certain value.

What students already know

- *Syntactic level + viable conceptual model*
  - Importing functions from libraries
  - Variables and assignments
  - Boolean expressions
  - `if` statement
  - `for` loop with explicit, automatically handled index ("true" definite iteration; e.g., Python `for i in range(1,11)`, Snap! (and Pascal) `for i:=1 to 10`, but not the 3-clause `for` of C and Java)
  - Function to generate a random integer in an interval

continued on next page

- *What students already know how to do (strategic level)*
  - Use a variable as a counter
  - Combine definite iteration with selection to iterate over a collection and perform an action on the basis of a condition

Target concept

- `while` loop (syntactic + conceptual)
- Repeating until a condition is met (without knowing how many iterations will take) (strategic)

Before the necessity sequence

Example of a previous task not needing the target concept

Count how many times a program that generates K pseudo-random integers between 1 and 1000 produces the number 42.

P!S – Problem-solving phase (unsolvable problem)

Problem

Count how many pseudo-random integers between 1 and 1000 a program generates before getting the number 42.

Necessity trigger

Students do not know a-priori how many calls the program needs to get 42.

Necessity

I need a way to repeat "until something happens" without knowing in advance when (or even if) it will happen.

Sub-optimal solutions

- (incomplete) Using a definite iteration to repeat a very high number of times (ideally to the MAXINT, if available in the language), hoping that 42 will be generated before.
- (complete) Using the (often not simple nor elegant) possibility of modern languages' `for`-like loops to realise an indefinite iteration.

I – Instruction phase

Illustrating the concept of indefinite repetition using the construct that repeats a block of code "while a condition <C> is True" realised in Python.

```
while <C>:
    <block of code>
```

continued on next page

PS – Second problem-solving phase
Target code

```
from random import randint
count = 1
while randint(1,1000) != 42
    count = count + 1
print(count)
```

Characteristics

- *Control* abstraction
- *Downward* abstraction movement (compared to the for loop with explicit, automatically handled index)
- *Hard* necessity (without the target concept, using (true) definite iteration, only incomplete solutions are possible; complete solutions are possible by using—often inelegantly hence sub-optimally—`for`-like loops to realise indefinite iteration)

Example discussion
The necessity that this example aims to stimulate would also be satisfied by using the recursive mechanism. Consequently, the same sequence can be used to introduce recursion.

Another essential clarification concerns a limitation of this *necessity* sequence. It can only be used with languages (such as Scratch and Python) in which the `for` loop realises a "true"[14] definite iteration. For obvious reasons, it cannot be used with languages in which the `for` construct (such as the 3-clause `for` of C and Java) can express indefinite iterations and thus be used instead of the `while` loop.

An alternative but equally viable scenario to stimulate this necessity is the well-known *Rainfall problem* (Soloway, 1986). The condition on which the program waits is not related to the extraction of a pseudo-random number but the user's input of a termination value. Only few prerequisites change: students need to know how to use basic input functionalities, they do not need to know how to import modules nor generate pseudo-random integers.

Another "less straightforward" way to stimulate the necessity of indefinite iteration is asking to terminate the scan before the end of the sequence (without using `break`, `return`, or auxiliary functions) when a certain condition is satisfied. For example, the problem might ask students to determine whether a very long sequence contains at least one certain integer and to do so in the shortest possible time. Consequently, since the sequence is very long, it is critical to stop as soon as the number is found. We consider this

[14] Please, recall the discussion about this matter in 3.2.3.

to be a more artificial request and consequently less effective in stimulating the necessity of indefinite iteration: this is not a problem that cannot be solved without the target concept, but that can be solved more efficiently with it.

4.4.3. *The necessity of arrays*

Necessity example 3

The necessity of arrays

Problem in a nutshell
Keeping track of how many times each number is drawn, with many possible numbers and many extractions.

What students already know

- *Syntactic level + viable conceptual model*
  - Variables and assignments
  - Boolean expressions
  - `for` statement
  - `if` statement
  - Function to generate a random integer in an interval
  - Function to plot a bar chart
- *What students already know how to do (strategic level)*
  - Using a variable as a counter

Target concept

- Arrays/lists with index access (syntactic + conceptual)
- Using arrays/lists to store, read and modify values accessing them by index (strategic)

Before the necessity sequence
Examples of previous tasks not needing the target concept

A. Given the possibility of tossing a coin (i.e., generating a random integer between 0 and 1), check whether the number of heads and tails is approximately equal after a high number of tosses (e.g., 10000).
B. Given the possibility of throwing a die (i.e., generating a random integer between 0 and 5), check whether the results of the throws are evenly distributed after a very high number of throws (e.g., 1 million).

continued on next page

P!S – Problem-solving phase (unsolvable problem)

Problem

Given a simplified version of the Bingo game (i.e., the possibility of drawing a number between 0 and 89), check whether the distribution of the results of the extractions is uniformly distributed through a very high number of extractions (e.g., 1 million).

Necessity trigger

Students need to create an unnaturally high number of unrelated variables and handle them with a very long sequence of selection statements, resulting in a program that is hard to manage without errors.

Necessity

I need a data structure to collect the frequency of extraction of each number and access (and modify) the values in that structure in a programmatic way based on runtime informations (i.e., the current number extracted).

Sub-optimal solution

- (complete) Create a variable for each number (90 variables) and, after every extraction, increment the corresponding variable through a multiple selection statement.

I – Instruction phase

Illustrating:

- how to create a (fixed length) ordered structure (array/list) and initialize it with constant values;
- how to access elements of the structure by integer index;
- how to modify an element by using index access on the left hand side of the assignment.

PS – Second problem-solving phase

Target code

An example, using Python lists as arrays, i.e., with fixed length.

```
from matplotlib.pyplot import *
from random import randint
L=[0]*90
for k in range(10**6):
    L[randint(0,89)] += 1

bar(range(90),L)
show()
```

continued on next page

Characteristics

- *Data* abstraction[a]
- *Upward* abstraction movement (compared to using many unrelated variables)
- *Soft* necessity (complete, though sub-optimal, solutions are possible without the target concept)

Implementation warning

Other concepts that students should not know yet for the mechanism to work:

- Dictionary-like data structures

[a] This data abstraction allows also for a powerful control abstraction. See the related discussion after the example.

Example discussion

This example concerns *data* abstraction because a set of unrelated variables is gathered in a single data structure. However, accessing by index to the elements of this structure allows a powerful *control* abstraction since it makes a long cascade of `if` statements (previously used to decide which variable to modify) disappear in one fell swoop.

Moreover, although the example uses Python's lists (i.e., dynamically extendable structures), the core necessity here is access by index. So this example is more generally suitable for use with languages providing array-like structures.

Lastly, the scenario in this example could be used to introduce dictionaries instead of arrays. However, we present below example 4 that we consider more effective for introducing dictionaries since the *necessity* problem requires the use of non-numeric indexes.

4.4.4. *The necessity of dictionaries*

Necessity example 4

The necessity of dictionaries

Problem in a nutshell

Counting the frequency of each character in an input string.

What students already know

- *Syntactic level + viable conceptual model*
  - Variables and assignments
  - `if` statement
  - Taking string input
  - Type conversion from `char` to `int`
  - `foreach` over a string
  - Arrays/lists with index access

continued on next page

- *What students already know how to do (strategic level)*
  - Check if a character is in a string
  - Use a variable as a counter
  - Iterating over values of a sequence
  - Combine definite iteration with selection to iterate over a collection and perform an action on the basis of a condition
  - Using arrays/lists to store, read and modify values accessing them by index

Target concept

- Dictionaries (syntactic + conceptual)
- Using dictionaries to store, read and modify values accessing them by key (strategic)

Before the necessity sequence

Example of a previous task not needing the target concept

Given a string taken from user input, determine the frequency of each of the ten digits in the string.

P!S – Problem-solving phase (unsolvable problem)

Problem

Given a string taken from user input in an unknown (potentially very long) alphabet, determine the frequency of each character in the string.

Necessity trigger

Students need to keep correspondence between each character and its frequency. However, arrays allow accessing values only by integer indexes.

Necessity

I need a way to associate a non-integer element (i.e., a character) with its frequency and use that element as a *key* to access and modify such frequency.

Sub-optimal solution

(complete) Creating two "parallel" arrays/lists. The first one stores each character encountered, and the second one stores the frequency of that character in the corresponding position.
Note that a way to dynamically increase the structures' length is needed because it is unknown beforehand how many different characters are in the string.
Note also that looking up for characters (and their corresponding integer index) in the first array/list introduces a high computational overhead[a].

continued on next page

I – Instruction phase

Illustrating:

- how to create a dictionary[b] data structure that keeps a collection of (key,value) pairs, i.e., a correspondence between unique keys and arbitrary values;
- check if a key is already associated with a value in a dictionary;
- how to access dictionary elements by key;
- how to modify a dictionary element by using key access on the left-hand side of the assignment.

PS – Second problem-solving phase

Target code

We propose a solution using Python dictionaries.

```
text = input()
freq = {}
for c in text:
    if c not in freq:
        freq[c] = 1
    else:
        freq[c] += 1
```

Characteristics

- *Data* abstraction
- *Upward* abstraction movement (compared to using structures with only integer index access)
- *Hard* necessity (although complete but still heavily sub-optimal solutions are possible without the target concept, they require advanced concepts such as parallel arrays and dynamically increasing structures)

[a] For each character `c` of the input string, $O($`len(L_CAR)`$)$ to determine if `c` is already present in `L_CAR` and, if present, to determine its integer index.

[b] Often called *associative array*.

Example discussion

Although the presented sub-optimal solution (using parallel lists) is complete, we still consider this to be an *educationally hard necessity*. Indeed, if students are not aware of the parallel arrays/lists pattern, it is unlikely that they can conceive this solution strategy.

Furthermore, we believe it is best if students are not exposed to the use of arrays/-parallel lists to keep a correspondence between different sets of elements in order to increase the efficacy of this necessity sequence. Because this programming pattern allows

the problem to be solved completely and quite easily (though less straightforwardly than using dictionaries), the necessity perceived by students would be much weaker. Besides becoming a *soft necessity* scenario (since students could resort to the tools they already know, see 3.2.3), the feeling of necessity would concern not the possibility to solve the problem but just the solution efficiency (significant and observable only with very long inputs) and the conciseness of the code.

We make this remark because, traditionally, programming courses teach this pattern. However, in languages that include more advanced data abstractions (such as dictionaries or OOP), we think it is not advisable to teach it, as it is less efficient, concise and elegant than its more advanced alternatives.

5. Conclusions

Two are the main contributions of this paper.

1. The proposal of a specific learning design for CS1 programming (NLD), based on the *necessity mechanism*.
   The learning design asks students to engage with problems that are very similar to those they already successfully faced, but this time they miss an essential ingredient (the target concept). Hence, struggling to solve the problem without success, they will experience the *necessity* of that concept.
2. A series of examples, framed in a concrete CS1 path, of NLD use in specific learning moments when abstraction changes.
   These moments are the ups and downs of what we call the *rollercoaster of abstraction*. Research shows that moving between the different levels of abstraction of programming language constructs is difficult for novices, both going upward and downward (albeit for different reasons).

Necessity learning design for CS1 programming (NLD). From an educational point of view, our design is inspired by PS-I approaches and, in particular, by Productive Failure learning design. However, *necessity* learning is *domain-specific* (Nelson and Ko, 2018) since it leverages inherent aspects of programming problems, like programs interactivity and the possibility to have honest feedback from the machine to check whether the problem is solved or not. Therefore, we developed a three-phase approach, synthetically described with P!S-I-PS.

i. In the **P!S** phase, students are not able to solve (or optimally solve) a given programming problem, experiencing the *necessity* of the target concept.
ii. In the **I** phase, unlike in Productive Failure, students are not given the solution. The target concept and its general usage are directly taught.
iii. In the final **PS** phase, students go back to the problem with the necessary knowledge to solve it, building on their previous failed attempts.

Moreover, this approach seems more generally in line with the vast body of research on CS1 courses for various reasons.

- It falls in the domain of active-learning designs since students are actively involved in solving a problem.
- It can help reduce the cognitive load since it allows for a very gradual path, in which new concepts are a *minimal addition* to the previous knowledge and are introduced in isolated situations.
- Problems are carefully crafted to capture the essence of the target concept, i.e., meaningful, prototypical examples of the use of that concept.
- Problems are built so that the target concept is (at least at that specific point in the learning path) the optimal one to use to solve that problem. Therefore, our design helps not to focus only on syntactical (and conceptual) knowledge but is especially useful for fostering strategic knowledge since it puts students right away in a situation where the target concept is essential to reach the *purpose*.
- The examples we presented are mostly built around the *edges* of basic programming concepts. This choice is in line with the *Learning Edge Momentum* hypothesis, suggesting that paying careful attention to introducing the basic concepts and their connections is essential to avoid a *negative momentum*. We argue that our approach can help those students who typically fail to keep the pace of CS1 courses from early stages.

NLD examples in the abstraction rollercoaster. We proposed examples of NLD use in CS1 programming learning moments in which the level of abstraction changes because the literature recognises these moments as critical. Indeed, we recognised a "rollercoaster" that goes up and down the level of abstraction within the programming language chosen for CS1. Both directions are challenging for novices, require different kinds of attention from educators and present different scenarios for the use of NLD.

- Going down in abstraction is problematic because it adds details, so the cognitive load increases. Usually, to stimulate the right necessity in students, they have to be put in a situation where those extra details are necessary to solve the problem.
- Going up is also tricky because details that students have learned to control and master are taken away. Teachers have to convince students that the new construct makes their life easier because it is either more simple, efficient, elegant or expressive (or a combination of these).

Although we acknowledge that there is no agreement among researchers and educators on the pathway to follow in CS1, we proposed a concrete learning path (greatly based on the CS1 for Math major successfully tested in years, also online) and four *necessity* sequences (the examples in 4.4) in it. As extensively discussed, the path (particularly the order of topics) determines the abstraction movements in it and thus influences the choice and design of *necessity* sequences.

Future developments. Our proposal is grounded in the literature, and we believe it is a sound *necessity-driven* learning design. We informally experimented with it in several

editions of a CS1 for Math major at the undergraduate level. More controlled testing should be done to obtain quantitative and qualitative measures of its effectiveness and impact on aspects like cognitive load, different types of programming knowledge, learning edge momentum.

We would be delighted to receive from educators examples of *necessity* situations they recognise in their teaching (contextualised in the learning path, to identify the direction of the abstraction movement) to help us build relevant *necessity* sequences so as to cover more and more learning moments in different CS1 paths and continue to refine our learning design.

References

Alexandron, G., Armoni, M., Gordon, M., Harel, D. (2012). The Effect of Previous Programming Experience on the Learning of Scenario-Based Programming. In: *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. Koli Calling '12. Association for Computing Machinery, New York, NY, USA, pp. 151–159. 9781450317955. https://doi.org/10.1145/2401796.2401821.

Barrows, H.S. (1996). Problem-based learning in medicine and beyond: A brief overview. *New Directions for Teaching and Learning*, 1996(68), 3–12. https://doi.org/10.1002/tl.37219966804.

Bawamohiddin, A.B., Razali, R. (2017). Problem-based Learning for Programming Education. *International Journal on Advanced Science, Engineering and Information Technology*, 7(6), 2035. https://doi.org/10.18517/ijaseit.7.6.2232.

Bayman, P., Mayer, R.E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology*, 80(3), 291–298. https://doi.org/10.1037/0022-0663.80.3.291.

Bjork, E., Bjork, R. (2011). Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society*, 56–64.

Caspersen, M.E. (2018). Teaching Programming. In: Sentance, S., Barendsen, E., Schulte, C. (Eds.), *Computer science education: perspectives on teaching and learning in school*. Bloomsbury Academic, London-New York, pp. 109–130. 9781350057111 9781350057104.

Colburn, T., Shute, G. (2007). Abstraction in Computer Science. *Minds and Machines*, 17(2), 169–184. https://doi.org/10.1007/s11023-007-9061-7.

Costantini, U., Lonati, V., Morpurgo, A. (2020). How Plans Occur in Novices' Programs: A Method to Evaluate Program-Writing Skills. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Association for Computing Machinery, New York, NY, USA, pp. 852–858. 9781450367936. https://doi.org/10.1145/3328778.3366870.

Curzon, P., Bell, T., Waite, J., Dorling, M. (2019). Computational Thinking. In: Fincher, S.A., Robins, A.V. (Eds.), *The Cambridge Handbook of Computing Education Research*. Cambridge Handbooks in Psychology. Cambridge University Press, Cambridge, pp. 513–546. https://doi.org/10.1017/9781108654555.018.

Deek, F., Kimmel, H., McHugh, J.A. (1998). Pedagogical Changes in the Delivery of the First-Course in Computer Science: Problem Solving, Then Programming. *Journal of Engineering Education*, 87(3), 313–320. https://doi.org/10.1002/j.2168-9830.1998.tb00359.x.

Downey, A. (2015). *Think Python*. O'Reilly Media, Sebastopol, CA. 978-1491939369.

Falkner, K., Sheard, J. (2019). Pedagogic Approaches. In: Fincher, S.A., Robins, A.V. (Eds.), *The Cambridge Handbook of Computing Education Research*. Cambridge Handbooks in Psychology. Cambridge University Press, Cambridge, pp. 445–480. https://doi.org/10.1017/9781108654555.016.

Freeman, S., Eddy, S.L., McDonough, M., Smith, M.K., Okoroafor, N., Jordt, H., Wenderoth, M.P. (2014). Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences*, 111(23), 8410–8415. https://doi.org/10.1073/pnas.1319030111.

Gabbrielli, M., Martini, S. (2010). *Programming Languages: Principles and Paradigms*. Springer London, London. https://doi.org/10.1007/978-1-84882-914-5.

Glogger-Frey, I., Fleischer, C., Grüny, L., Kappich, J., Renkl, A. (2015). Inventing a solution and studying a worked solution prepare differently for learning from direct instruction. *Learning and Instruction*, 39. https://doi.org/10.1016/j.learninstruc.2015.05.001.

Guttag, J. (2021). *Introduction to Computation and Programming Using Python : With Application to Computational Modeling and Understanding Data*. The MIT Press, Cambridge, Massachusetts. 9780262542364.

Guzdial, M. (2008). Paving the Way for Computational Thinking. *Commun. ACM*, 51(8), 25–27. https://doi.org/10.1145/1378704.1378713.

Guzdial, M. (2017). Balancing Teaching CS Efficiently with Motivating Students. *Commun. ACM*, 60(6), 10–11. https://doi.org/10.1145/3077227.

Kapur, M. (2016). Examining Productive Failure, Productive Success, Unproductive Failure, and Unproductive Success in Learning. *Educational Psychologist*, 51(2), 289–299. https://doi.org/10.1080/00461520.2016.1155457.

Kapur, M., Bielaczyc, K. (2012). Designing for Productive Failure. *Journal of the Learning Sciences*, 21(1), 45–83. https://doi.org/10.1080/10508406.2011.591717.

Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J.H., Crawford, K. (2000). Problem-Based Learning for Foundation Computer Science Courses. *Computer Science Education*, 10(2), 109–128. https://doi.org/10.1076/0899-3408(200008)10:2;1-c;ft109.

Lodi, M. (2020). Informatical Thinking. *Olympiads In Informatics*, 14, 113–132. https://doi.org/10.15388/ioi.2020.09.

Lodi, M., Martini, S. (2021). Computational Thinking, Between Papert and Wing. *Science & Education*, 30(4). https://doi.org/10.1007/s11191-021-00202-5.

Lodi, M., Sbaraglia, M., Zingaro, S.P., Martini, S. (2021). The Online Course Was Great: I Would Attend It Face-to-Face: The Good, The Bad, and the Ugly of IT in Emergency Remote Teaching of CS1. In: *Proceedings of the Conference on Information Technology for Social Good*. GoodIT '21. Association for Computing Machinery, New York, NY, USA, pp. 242–247. https://doi.org/10.1145/3462203.3475902.

Loibl, K., Rummel, N. (2014). The impact of guidance during problem-solving prior to instruction on students' inventions and learning outcomes. *Instructional Science*, 42. https://doi.org/10.1007/s11251-013-9282-5.

Loibl, K., Roll, I., Rummel, N. (2017). Towards a Theory of When and How Problem Solving Followed by Instruction Supports Learning. *Educational Psychology Review*, 29(4), 693–715. https://doi.org/10.1007/s10648-016-9379-x.

Martini, S. (2016a). Several types of types in programming languages. In: Gadducci, F., Tavosanis, M. (Eds.), *HAPOC 2015*. IFIP Advances in Information and Communication Technology. Springer, Cham, pp. 216–227. https://doi.org/10.1007/978-3-319-47286-7_15.

Martini, S. (2016b). Types in Programming Languages, between Modelling, Abstraction, and Correctness. In: Beckmann, A., Bienvenu, L., Jonoska, N. (Eds.), *CiE 2016: Pursuit of the Universal*. LNCS*: Vol.* 9709. Springer, Cham, pp. 164–169. https://doi.org/10.1007/978-3-319-40189-8_17.

Martini, S. (2020). The Standard Model for Programming Languages: The Birth of a Mathematical Theory of Computation. In: de Boer, F.S., Mauro, J. (Eds.), *Recent Developments in the Design and Implementation of Programming Languages*. OpenAccess Series in Informatics (OASIcs)*: Vol.* 86. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1–13. 978-3-95977-171-9. https://doi.org/10.4230/OASIcs.Gabbrielli.8.

McGill, T.J., Volet, S.E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education*, 29(3), 276–297. https://doi.org/10.1080/08886504.1997.10782199.

Moallem, M., Hung, W., Dabbagh, N. (Eds.) (2019). *The Wiley Handbook of Problem-Based Learning*. John Wiley & Sons, Hoboken, NJ. https://doi.org/10.1002/9781119173243.

Nelson, G.L., Ko, A.J. (2018). On Use of Theory in Computing Education Research. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, New York. https://doi.org/10.1145/3230977.3230992.

Nuutila, E., Törmä, S., Kinnunen, P., Malmi, L. (2008). Learning Programming with the PBL Method — Experiences on PBL Cases and Tutoring. In: Bennedsen, J., Caspersen, M.E., Kölling, M. (Eds.), *Reflections on the Teaching of Programming: Methods and Implementations*. Springer, Berlin, Heidelberg, pp. 47–67. 978-3-540-77934-6. https://doi.org/10.1007/978-3-540-77934-6_5.

O'Grady, M.J. (2012). Practical Problem-Based Learning in Computing Education. *ACM Transactions on Computing Education*, 12(3), 1–16. https://doi.org/10.1145/2275597.2275599.

Oliveira, A.M.C.A., dos Santos, S.C., Garcia, V.C. (2013). PBL in teaching computing: An overview of the last 15 years. In: *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE, Oklahoma City, Oklahoma. https://doi.org/10.1109/fie.2013.6684830.

Pane, J.F., Ratanamahatana, C., Myers, B.A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2), 237–264. https://doi.org/10.1006/ijhc.2000.0410.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York. 0-465-04627-4.

Peng, W. (2010). Practice and experience in the application of problem-based learning in computer programming course. In: *2010 International Conference on Educational and Information Technology*. IEEE, Chongqing, China. https://doi.org/10.1109/iceit.2010.5607778.

Prince, M. (2004). Does Active Learning Work? A Review of the Research. *Journal of Engineering Education*, 93(3), 223–231. https://doi.org/10.1002/j.2168-9830.2004.tb00809.x.

Robins, A. (2010). Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, 20(1), 37–71. https://doi.org/10.1080/08993401003612167.

Robins, A.V. (2019). Novice Programmers and Introductory Programming. In: Fincher, S.A., Robins, A.V.E. (Eds.), *The Cambridge Handbook of Computing Education Research*. Cambridge Handbooks in Psychology. Cambridge University Press, Cambridge, pp. 327–376. https://doi.org/10.1017/9781108654555.013.

Schwartz, D., Martin, T. (2004). Inventing to Prepare for Future Learning: The Hidden Efficiency of Encouraging Original Student Production in Statistics Instruction. *Cognition and Instruction - COGNITION INSTRUCT*, 22, 129–184. https://doi.org/10.1207/s1532690xci2202_1.

Schwartz, D.L., Bransford, J.D. (1998). A Time For Telling. *Cognition and Instruction*, 16(4), 475–5223. https://doi.org/10.1207/s1532690xci1604_4.

Shneiderman, B. (1977). Teaching programming: A spiral approach to syntax and semantics. *Computers & Education*, 1(4), 193–197. https://doi.org/10.1016/0360-1315(77)90008-2.

Sinha, T., Kapur, M. (2019). When productive failure fails. *Europe (Germany, Switzerland, UK)*, 30, 31–6.

Sinha, T., Kapur, M. (2021). Robust Effects of the Efficacy of Explicit Failure-driven Scaffolding in Problem-Solving Prior to Instruction: A Replication and Extension. *Learning and Instruction*, 75, 101488. https://doi.org/10.1016/j.learninstruc.2021.101488.

Soloway, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations, *29*(9), 850–858. https://doi.org/10.1145/6592.6594.

Statter, D., Armoni, M. (2020). Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Trans. Comput. Educ.*, 20(1). https://doi.org/10.1145/3372143.

Turner, R. (2021). Computational Abstraction. *Entropy*, 23(2). https://doi.org/10.3390/e23020213.

VanLehn, K., Siler, S., Murray, R.C., Yamauchi, T., Baggett, W. (2003). Why Do Only Some Events Cause Learning During Human Tutoring? *Cognition and Instruction - COGNITION INSTRUCT*, 21, 209–249. https://doi.org/10.1207/S1532690XCI2103_01.

Visser, E. (2015). Understanding software through linguistic abstraction. *Science of Computer Programming*, 97, 11–16. Special Issue on New Ideas and Emerging Results in Understanding Software. https://doi.org/10.1016/j.scico.2013.12.001.

M. Sbaraglia holds bachelor and master degrees in Informatics Engineering. After researching biochemically inspired self-organizing systems (publishing in an international journal), he worked for a few years as a developer. He has been teaching Informatics in high school since 2013 and has been holding a chair since 2016. Driven by an interest in Informatics Education, he is now a PhD student in Informatics (at Alma Mater Studiorum - Università di Bologna, Italy). He researches introductory programming, CS1 online learning, big ideas of Cryptography and interdisciplinarity between Informatics and Mathematics. He published in international conferences and journals of Informatics and Science Education.

M. Lodi holds bachelor, master and PhD degrees in Informatics. He is currently a post-doctoral researcher and adjunct professor of Informatics Education at Alma Mater Studiorum - Università di Bologna, Italy. His research interest is in informatics education, particularly on computational thinking with a constructivist and constructionist approach, teaching of cryptography, transfer of learning, informatics mindset, and epistemological aspects of informatics as a discipline. He is the author of more than fifteen publications in international conferences and journals on informatics education and a book in Italian for primary school teachers. He is actively involved in national initiatives to introduce Informatics in the Italian K-12 curriculum. https://lodi.ml

S. Martini (Ph.D. in Computer Science, University of Pisa, 1987) is professor of Computer Science at Alma Mater Studiorum – Università di Bologna, Italy. Before joining Università di Bologna in 2002, he taught at the universities of Pisa and Udine. He has been a visiting scientist at the former Systems Research Center of Digital Equipment Corporation, Palo Alto; at Stanford University; at École normale supérieure, Paris; at Université Paris 13; at University of California at Santa Cruz; and the Collegium - Lyon Institute for Advanced Studies. His research interests are in the logical foundations of programming languages, in history and philosophy of computer science, and in computer science education.