Informatics in Education, 2026, Vol. 0, No. 0, 1–31

DOI: https://doi.org/10.15388/infedu.2604.035

A Cognitive-Load-Informed Decomposition of Debugging Subskills With Targeted Exercises

Gabriele POZZAN[1,*] *[0000-0002-3575-6233]*,
Andreas BOLLIN[2] *[0000-0003-4031-5982]*,
Tullio VARDANEGA[1] *[0000-0002-0089-0889]*

[1] *Department of Mathematics, University of Padua, Italy*
[2] *Department of Informatics Didactics, University of Klagenfurt, Austria*
e-mail: gabriele.pozzan@unipd.it, Andreas.Bollin@aau.at, tullio.vardanega@unipd.it

Received: 25 April 2026

Abstract. Debugging is integral to programming. It comes into play as soon as novices make their first mistakes in creating programming artifacts. It is also consistently reported to be a skill that is difficult to learn as well as to teach effectively. Research in Informatics Education has often focused on the *process* of debugging, by breaking it down in steps connected by temporal and causal dependencies. In this work, we focus instead on debugging *as a skill*, from the standpoint of Cognitive Load Theory, and break it down into a tree-shaped model of subskills that enable one another. Debugging may thus be seen as a *meta*-skill that requires the coordination of multiple others. From the standpoint of Cognitive Load Theory, such a skill is cognitively expensive, which may explain the learning-related difficulties tied to debugging. Using the framework of the four-component instructional design, we hypothesize a categorization of each debugging subskill as either recurrent or nonrecurrent, dividing those that are applied consistently to different contexts from those that require problem solving. All subskills may be practised and potentially assessed with targeted exercises, whose design depends on their recurrent/nonrecurrent nature. We provide extensive examples of such exercises. Our decomposition of debugging into subskills is a novel way to address debugging in educational contexts and complements the work done on debugging processes. Although it is currently a theoretically grounded conjecture, the model provides concrete guidance for instructors on analyzing existing materials and planning cognitive-load-informed learning trajectories.

Key words: debugging, cognitive load, 4C/ID, learning tasks.

1. Introduction

Debugging – the back-and-forth process of diagnosing and fixing programming bugs – is inextricably tied to the act of programming. This is especially true in learning contexts: as soon as novice programmers start creating programming

*Corresponding author: Gabriele POZZAN.

artifacts, they will inevitably make errors that may result in faults that may in turn manifest as failures, viz. outcomes that deviate from the intended specifications. The literature on debugging in learning contexts is rich and dates back to the 1970s. However, some of the conclusions of this vast body of work are still relevant today, as Informatics is becoming more ubiquitous in school curricula at all levels, but this expansion is not always matched by corresponding preparation in teacher education. For example: the 2022 "Informatics education at school in Europe" analysis (European Commission/EACEA/Eurydice, 2022) reports that, among 39 education systems in Europe, 23 teach Informatics as a distinct discipline at the primary school level, but only 3 require primary school Informatics teachers to be specialized in the discipline.

A 2008 review by McCauley et al. (2008) reports that debugging is often both difficult to learn for students and difficult to teach for instructors. This notion is corroborated by a more recent analysis by Yang et al. (2024). Both surveys maintain that debugging skills are not a direct consequence of programming skills. Experimental studies, such as the analyses of Ahmadzadeh et al. (2005) and Fitzgerald et al. (2008), support this notion: both works examine the correlations between learners' programming and debugging skills and find that, while "good programmers" are not necessarily "good debuggers", the converse is not true: "good debuggers" tend indeed to be also "good programmers". This observation supports the argument that systematic, explicit teaching of debugging could and should be done starting from introductory programming courses.

A 2024 meta-analysis by Sun et al. (2024), focusing on interventions aimed at improving debugging skills, corroborates this conclusion. One of the conclusions of the analysis is that studies focused on systematic debugging instruction have the overall second-best results, closely following those based on enhanced debuggers. Interestingly, the authors note that debugging involves the *coordination* of different subskills, e.g., program comprehension, hypothesis testing, etc.

The present work is based on the same premise: we model the *decomposition of debugging in subskills*. As this is a theoretical work, our model is meant to lay foundations for future research in the teaching and learning of debugging. The model is inspired by the Cognitive Load Theory (CLT) pioneered by J. Sweller (Sweller et al., 2019) and by the work of J. van Merriënboer (van Merriënboer et al., 2002) on the four-component instructional design (4C/ID). We present a tree-shaped model of debugging subskills and show how they relate to one another. In modeling these relationships, we are interested in capturing how the subskills enable – i.e., are prerequisite to – one another.

In this work, we are not primarily interested in temporal dependencies, which are the focus of debugging *process* models, discussed in depth in the literature and summarized here in Section 2.4. Debugging process models describe debugging as a sequence of actions, this work instead describes it as a dependency structure of subskills that impose cognitive load (CL) on learners.

The main conceptual contribution of this work is a *coherent model* of debugging that *systematizes scientific sources* discussing its subskills. To the best of

our knowledge, this particular educational perspective on debugging has not been explored yet in Informatics Education research. The purpose of the model is to support instructional diagnosis, sequencing, and exercise design. The presentation of our model is therefore accompanied by extensive examples of exercises that may be used to support practice and assessment of specific subskills of debugging, detailed in Section 4. We exclude debugging-related activities such as bug reporting and documentation: while important in professional contexts, they are not key for novice learning.

Our decomposition is broadly applicable to Informatics Education. It may be especially useful as support in the design of introductory programming courses (CS1) at the higher-secondary or university level: the model suggests viable sequencing of learning goals and activities that should build strong foundations for debugging. As a case study, throughout this paper, we discuss the application of the model in the context of preparation for competitive programming events, such as the International Olympiad in Informatics.[2] Contest environments impose strict constraints (time/memory limits, I/O specifications, restricted toolchains) and judge feedback (see Section 3.4) that require efficient coordination of debugging subskills under cognitive pressure. Our model helps instructors sequence instruction and practice to manage intrinsic and extraneous CL, identify subskills that underlie typical contest failures (e.g., runtime errors via out-of-bounds access or overflow), and design part-task drills that build fluency before full contest-style problem solving.

In the following, we use the terminology established by the IEEE Standard Classification for Software Anomalies (IEEE Computer Society, 2010):

- **Error**: an action that leads to a result – e.g., a program – different from the expected/desired – e.g., the program may not compile, it may not meet the requirements, etc.
- **Fault**: the result of an error, manifested as some feature – or lack thereof – of a software program.
- **Bug**: a synonym of "Fault", often used with the connotation of *being present in a finished/shipped product*. We shall prefer this term, as it clearly captures the meaning of "being the object of *debugging*".
- **Failure**: the deviation of the behavior of a program from its specification and requirements. This is the surface level, evident, manifestation of an underlying bug/fault.
- **Debugging**: the skill/activity of diagnosing and fixing faults/bugs or, in other words, making programs converge towards their intended behavior.

The remainder of this paper is organized as follows. Section 2 discusses related work that forms the theoretical background of our model. Section 3 presents our debugging decomposition model. Section 4 provides examples of exercises for

[2] https://ioinformatics.org/

specific debugging subskills. Section 5 contains a discussion of our results, separating what we conclude from what we may only hypothesize. Section 6 draws conclusions on this work.

2. Related Work

2.1. *Cognitive Load Theory*

Our debugging decomposition model is based on Cognitive Load Theory (CLT). In this Section, we summarize its main concepts. A detailed discussion of the many facets of CLT may be found in recent work by Sweller et al. (2019).

CLT structures the cognitive architecture related to the learning and recalling of information around two types of memory. **Long-term memory** (LTM) is the virtually unlimited repository of all information that a human may take in as personal "knowledge". Knowledge in LTM is *schematized*, i.e., new information is stored in relation to existing schemas, rather than in isolation. **Working memory** (WM) is the moment-to-moment blackboard of conscious thought, that is, the temporary storage for things that a human can think about *instantaneously*. WM is the interface to both novel information and to the schemas stored in LTM. Schemas are not accessed directly but via the WM, where they must be loaded first. There is a strict limit to the amount of schemas and novel pieces of information that may be held in WM at the same time. Moreover, novel information tends to be evicted quickly, unless stored in LTM. Conversely, as schemas aggregate together complex, interconnected knowledge, they allow large amounts of related information to be "loaded" in WM as a single unit, resulting in CL that is vastly *less* than the sum of its parts. This is the mechanism that allows humans to process complex knowledge through the strict bottleneck of WM.

When encountering a new concept, the CL placed on WM is determined by two components:

- **Intrinsic CL** is related to both the complexity of the concept itself and the previous knowledge of the learner. In educational contexts, reducing intrinsic CL involves reducing the complexity of the learning materials, e.g., by decomposing complex concepts in their constituent parts. Moreover, the intrinsic CL of a topic naturally decreases as learners' mastery of it grows. This is highly relevant to the current work, as using exercises to evaluate expertise on specific subskills of debugging may be an indirect source of information on how such skills affect learners' CL at a given time.
- **Extraneous CL** is related to how concepts are presented. In educational contexts, it may be reduced by carefully designing learning materials in order to avoid superfluous information, ambiguity, and, in general, sources of distraction.

Being aware of the interplay of different subskills in debugging activities may be useful to reduce extraneous CL: e.g., by scaffolding parts of the learning activities

related to subskills that are not the primary focus of the current learning goals. Decomposing debugging into constituent subskills may also help instructors to both recognize and manage the intrinsic CL related to this complex meta-skill. Finally, structuring a learning trajectory around the dependencies of our model may help distribute intrinsic CL throughout the entire curriculum, thus lowering its peaks.

2.2. *Four-Component Instructional Design*

The second cornerstone of our approach is the 4C/ID developed by van Merriënboer et al. (2002). Of particular interest to our model is the 4C/ID categorization of skills based on their expected *outcomes*.

Recurrent skills are applied in different contexts in highly consistent ways. For example, tracing a piece of code is the same activity, regardless of whether one is studying a design pattern or debugging a program. Recurrent skills' outcomes are comparable across different situations. This consistency allows experts to apply them mostly *automatically*, thus freeing CL capacity for more complex and nonrecurrent aspects of tasks. Mastering recurrent skills requires prolonged and deliberate practice in which the presence of immediate corrective feedback is key. Accordingly, the 4C/ID model includes *part-task practice* sessions, which consist of small, drill-type exercises aimed at strengthening recurrent skills.

Nonrecurrent skills produce outcomes that are highly dependent on context. They always involve a measure of problem solving and may be the result of coordinating a number of simpler constituent skills. Debugging may thus be viewed as a very high-level nonrecurrent skill. Nonrecurrent skills are mastered by generalization over different contexts of application. Thus, the 4C/ID includes *learning tasks*, which should be whole-task activities that require the complex coordination of multiple skills. The model emphasizes task variability as a basis for generalization, and supportive information, i.e., learning materials aimed at helping students build cognitive schemas and apply metacognition.

Our debugging decomposition model differentiates recurrent and nonrecurrent subskills. This categorization, in conjunction with the indications of the 4C/ID, may suggest how to best structure learning activities aimed at strengthening particular debugging subskills.

2.3. *Tracing*

Tracing, i.e., reading and executing – whether mentally or on paper – a piece of code, shapes one of the main branches of our model. We share the emphasis on this precursor skill with the research of the BRACElet group that culminated in Lister's three-stage model of the mental development of novice programmers (Lister, 2021). According to this model, students start their programming learning journey in the **pre-tracing** stage. This involves being unable to consistently trace

code, and is characterized by a large number of misconceptions. In our model, we consider this stage to be the "default", i.e., the bare – but fertile – ground on which debugging subskills grow. The second stage of Lister's model is called the **tracing** stage: it involves being able to consistently trace code *starting from concrete values*, hence specific instances of execution. Learners at this stage are not able to reliably abstract properties of programs. Instead, they reason about code instruction by instruction, possibly with the aid of tracing tables to keep track of state changes. In our model, we call this *mechanical tracing*. The third stage is called **post-tracing**: learners at this level can discern abstract relationships among the different parts of a program. This may involve reasoning about programs in terms of both control flow and data flow, as well as being able to establish pre/postconditions and invariants for fragments of code. In our model, we call this *abstract tracing*.

Importantly, Lister suggests that it is only after reaching this final stage that learners really become receptive to the benefits of "writing lots and lots of code". This conclusion is consistent with how CLT and the 4C/ID describe novice learning. Programming assignments – in which students are required to design and implement algorithms to solve given requirements – correspond to 4C/ID *learning tasks*, whose variety induces generalization in problem solving skills. However, these complex tasks should be supported by *part-task practice* focused on their recurring aspects and prerequisites, such as tracing. The dependencies of our model delineate learning trajectories consistent with those suggested by Lister. In particular, we share the view that an initial focus on tracing should be foundational for both code production and debugging.

2.4. *Debugging Process Models*

Debugging process models are decompositions of the act of debugging into steps connected by *temporal* dependencies. These make sense from the point of view of CLT, as they manage the (high) intrinsic CL of debugging by splitting it into more manageable subtasks with clear inputs and outputs. This *divide-et-impera* approach is explicitly showcased in one of the oldest debugging "algorithms": the Wolf Fence algorithm for debugging (Gauss, 1982). This is based on the notion that a bug – the wolf – will have observable symptoms, i.e., failures or, in metaphor, howls. By "listening to the wolf's howls" at specific points of the program – e.g., by carefully planting telemetries, via `print` statements, assertions, etc. – it is thus possible to segment ("fence") the program in partitions in which the bug may or may not manifest. This reduces CL at each iteration, by performing a binary search for the bug on smaller and smaller partitions of the program.

Böttcher et al. (2016) adopt the "Wolf Fence" algorithm and make some of its implied parts more explicit. "Listening to the wolf's howls" is split into: 1) finding a location to partition the system/program; 2) formulating expectations on the program's state at that point; 3) verifying the expectations. The act of

"fencing" the program is split into "finding a location to subdivide it" and "writing a test case" after finding a bug. This model places great emphasis on the role of *expectations* and *verification* in debugging.

Other models exist that share the "Wolf Fence" algorithm's coarse level of granularity. Carver and Risinger (1987) describe an approach that requires dividing the program into partitions and verifying the presence of bugs in each partition separately. This model highlights the importance of *previous experience* in diagnosing a bug, e.g., by looking for it in the most likely subprograms. Interestingly, it also includes a *mechanical tracing* step ("read every command"), should other, more abstract, ways of finding the bug fail. Spinellis (2018) describes a model that makes explicit the important step of *reproducing the failure* and then making hypotheses and "experiments" to verify its cause.

A recent model that differs from the ones discussed so far is proposed by Michaeli and Romeike (2019). It explicitly breaks the debugging process into three iterative steps with clearly defined exit conditions. Step one requires verifying that the program compiles successfully and provides substeps aimed at diagnosing and solving compile-time errors. Step two requires verifying that the program runs without runtime failures. Step three involves the formulation of expectations, based on requirements, and their verification, in order to locate faults. The authors of this process model evaluated it by comparing a group receiving an intervention which included the explicit teaching of the model with a control group who performed "regular" debugging tasks, with no explicit process model. The authors found that the group receiving the intervention improved significantly in self-efficacy, with no significant improvement for the control group. Moreover, they found that, while the groups had comparable debugging performance before the interventions, the group adopting the process model achieved significantly better results in a post evaluation.

Finally, Li et al. (2019) discuss a four-step debugging process whose theoretical value is helpful in grounding some of our choices. They subdivide debugging processes in the following steps:

1. **Construct the problem space**: understanding both the *intended* and *actual* programs, i.e., what the program is *trying* to do and what it *actually does*.
2. **Identify fault symptoms**: observing discrepancies between the *intended* and *actual* behavior of a program.
3. **Diagnose the fault**: using the fault symptoms to pinpoint the location of a bug. This is where strategies such as the debugging processes discussed previously in this Section come into play.
4. **Generate and verify solutions**: fixing the bug and verifying that the *actual* program converged on the *intended* program.

The model we present in this work *complements* process decompositions: it is not so much concerned with the steps necessary to debug but with how the subskills that are coordinated while debugging relate to each other and with the

order in which they should be addressed in learning. For example, we share the emphasis on distinguishing between *intended* and *actual* programs; however, we focus on identifying the skills necessary for the two operations.

2.5. *Case Study: Contest Pedagogy and the Olympiad Context*

Pedagogy for the Olympiads in Informatics emphasizes algorithmic problem solving under explicit constraints formalized in the IOI syllabus (International Olympiad in Informatics Scientific Committee, 2025). Novice and intermediate learners' mental models of execution (referred to here as *notional machines*) play a central role in how they reason about these constraints in practice (Sorva, 2013). Likewise, careful interpretation of problem statements and input/output (I/O) specifications underpins the ability to identify boundary cases and formalize expectations about correct behavior. In general, debugging and its constituent subskills are part of the testing-refinement cycle that is necessary for the incremental improvement of algorithmic solutions (K-12 Computer Science Framework Steering Committee, 2016).

Training approaches documented in contest and competitive-programming pedagogy typically combine fluency-building activities (e.g., code reading, control- and data-flow reasoning, stepwise tracing) with full problems that require coordinating multiple skills such as specification interpretation, hypothesis generation, and localized repair. Laaksonen, in a recent guide to competitive programming (Laaksonen, 2024), stresses that, while the main skill tested in such contexts is the nonrecurrent ability to solve difficult algorithmic problems, other recurrent skills, e.g., being able to implement specific data structures in executable code, should be automated as much as possible. This resonates with considerations from CLT and with 4C/ID's emphasis on pairing part-task practice (to automate recurrent skills) with *varied* learning tasks. Laaksonen indeed mentions that, when preparing for competitive programming contests, one should focus on *quality over quantity* of learning tasks.

Finally, online-judge verdicts and guidance for contestants commonly frame feedback in terms of standard verdicts (see Section 3.4) and connect them to specific classes of errors – e.g., language/toolchain fluency for compilation errors, execution-model misconceptions for runtime errors, specification or corner-case issues for wrong answers, and algorithmic or data-structure mismatches for time/memory exceedances. These observations motivate targeted drills and structured scaffolding in contest preparation.

3. Debugging Decomposition Model

3.1. *How the proposed model was built*

The model presented in this Section stems from theoretical work by the authors. We first selected CLT and, in particular, the 4C/ID (van Merriënboer et al., 2002)

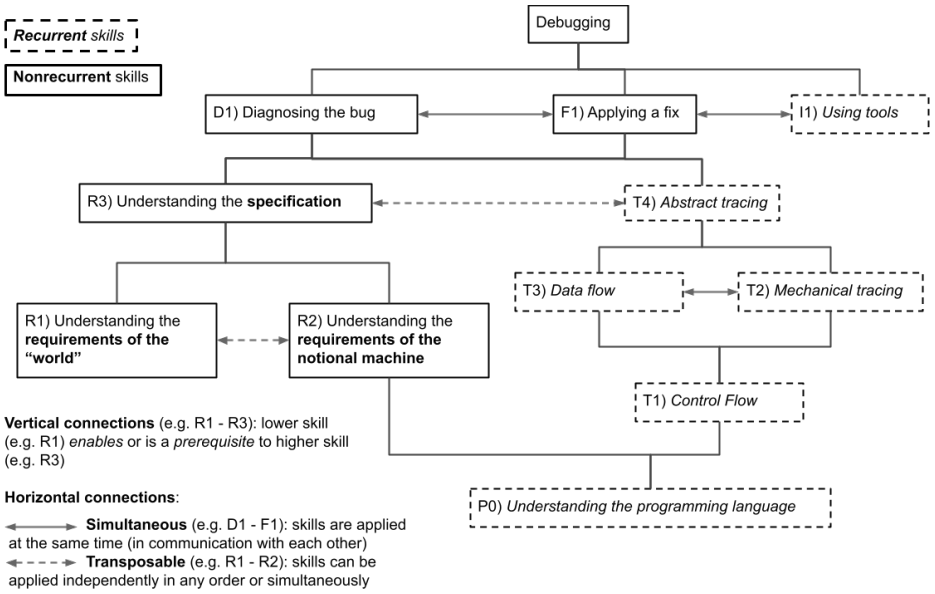

Figure 1. Debugging decomposition following the notation of van Merriënboer et al. (2002).

as the conceptual lens for analyzing and decomposing debugging as a meta-skill. The second foundation for our analysis is the framework for teaching debugging discussed by Li et al. (2019), specifically its discussion of the types of knowledge involved in debugging. Guided by these lenses, we organized the model into three sections:

1. The **debugging process**, motivated by the extensive literature on process models, summarized here in Section 2.4.
2. Understanding **the "intended" program**, drawing on the framework for debugging instruction by Li et al. (2019). As this branch relates to what a program aims to do – i.e., its *requirements* – and its *constraints*, we identified the work of Jackson (1995) as the main reference for its nodes and dependencies.
3. Understanding the **"actual" program**, again following Li et al. (2019). As this branch relates to what a program *actually does when executed*, we identified *tracing* as its main theme and consequently the work of the BRACElet group (Lister, 2021) as the main reference for its nodes and dependencies.

After identifying the nodes and dependencies of the graph, we categorized the subskills as recurrent/nonrecurrent, based on the distinctions made by van Merriënboer et al. (2002): recurrent skills are applied consistently from problem to problem and may be automatized to a certain degree, nonrecurrent skills require reasoning and problem solving. As the techniques related to tracing are highly transferable from context to context – e.g., different programming languages – we identified the subskills related to understanding the "actual" program as recurrent.

Id	Name	Source	Category	Enables	Enabled by
P0	Understanding the programming language	Li et al. (2019)	Recurrent	`T1` (C), `R2` (C)	
T1	Control Flow	Li et al. (2019)	Recurrent	`T2` (C), `T3` (C)	`P0` (C)
T2	Mechanical tracing	Lister (2021)	Recurrent	`T4` (S)	`T1` (C)
T3	Data flow	Li et al. (2019)	Recurrent	`T4` (C)	`T1` (C)
T4	Abstract tracing	Lister (2021)	Recurrent	`D1` (C), `F1` (C)	`T2` (S), `T3` (C)
R1	Understanding the requirements of the "world"	Jackson (1995)	Nonrecurrent	`R3` (S)	
R2	Understanding the requirements of the notional machine	Jackson (1995)	Nonrecurrent	`R3` (S)	`P0` (C)
R3	Understanding the specification	Jackson (1995)	Nonrecurrent	`D1` (C), `F1` (C)	`R1` (S), `R2` (S)
D1	Diagnosing the bug	Li et al. (2019)	Nonrecurrent		`R3` (C), `T4` (C)
F1	Applying a fix	Li et al. (2019)	Nonrecurrent		`R3` (C), `T4` (C)
I1	Using tools	Li et al. (2019)	Recurrent		

Table 1
Summary of the debugging subskills of the model depicted in Figure 1, their main sources in the scientific literature, their hypothesized recurrent/nonrecurrent categories, and their dependencies. The dependencies are labeled based on their being supported by prior work (S) or based on conceptual reasoning by the authors (C).

As understanding the "intended" program instead requires combining knowledge of the problem domain and of the notional machine – both context-specific – to understand the specification, we identified the related subskills as nonrecurrent. Diagnosing and fixing bugs are by definition problem solving activities, so we assigned them to the nonrecurrent category. Finally, as the additional CL imposed by using tools is related to learners' ability to operate them – which may be automated with experience – we assigned this last subskill to the recurrent category. Note that there may arguably be nuance on the recurrent/nonrecurrent nature of specific subskills (we discuss this for some of the exercises of Section 4) so these categorizations are best understood as solid suggestions that should help instructors in choosing/designing learning activities.

3.2. *Notation and Properties*

Figure 1 displays our model, whose main features are also summarized in Table 1. Its tree-like notation is based on the one used by van Merriënboer et al. (2002). Each rectangular node represents a debugging subskill. Recurrent subskills are highlighted in *italics* and surrounded by dashed boxes. Nonrecurrent subskills are enclosed by continuous boxes. Vertical connections, modeled by elbow connectors such as `R1-R3`, represent prerequisites: the skill that appears lower in the tree (`R1`) *enables*/is a *prerequisite* to the one that appears higher (`R3`). Horizontal connections, modeled by double arrows, represent temporal relationships. Continuous arrows (e.g. `T2-T3`) represent *simultaneous* relationships. Skills in such a relationship need to be applied at the same time – and possibly feed their outputs into each other as inputs – in order to enable other skills higher in the tree. Dashed arrows (e.g. `R1-R2`) represent *transposable* relationships between skills that may be applied in any order.

Educational activities that include a skill (e.g., `T3`) at any level may exclude the skills to which it is a prerequisite (e.g., `T4`, `F1`, etc.). For example, it is possible to focus on the control flow of a program without needing to fully trace it; it is also possible to understand the requirements/constraints of a notional machine without considering the full specification of a program, etc. However, it is not possible to *fully exclude* the skills that are themselves prerequisites to the focus of the educational activity (e.g., for `T3`, `T1` and `P0`), which may at best be simplified by scaffolding[3]. This implies that skills with longer lists of – transitive – prerequisites will likely induce higher intrinsic CL to learners.

3.3. *Model Details*

3.3.1. *Foundation: P0.*

Node `P0` – understanding the programming language – is one of the three leaves of our model. It is a foundational prerequisite to its two main branches. "Understanding" encompasses all the aspects of the *semiotics* of the language: syntax (rules of writing), semantics (meaning of constructs), pragmatics (execution effects), as discussed by Zemanek (1966). Li et al. (2019), categorize this subskill as part of the *domain knowledge* necessary for debugging. Note that learners need *not* understand *the entirety* of a programming language before being able to "climb" our debugging model. This is true in general for all the nodes of the graph: having mastered some concepts related to a particular subskill allows learners to use those concepts in a higher-level subskill that is dependent on it, and so forth. Unlike real-life climbing, the best approach to traverse our model is spiral-shaped,

[3] An apparent violation of this property would be diagnosing a bug in a procedure (`D1`) by finding a specific input that always leads to a failure. This approach *may* be carried out only with mechanical tracing (`T2`). However, the specific failure-inducing input will belong to a *class* of inputs, described by some feature that is the ultimate cause of the failure. The abstract description of this class of inputs belongs to `T4`.

with periodic revisiting of lower-level subskills at increasing levels of detail and complexity.

3.3.2. *The intended program: R1, R2, R3.*
The two main branches of our model are related to the important distinction that various works (Li et al., 2019; McCauley et al., 2008) report between understanding the *intended* vs. the *actual* program to debug. Nodes `R1`, `R2` and `R3` enable acquiring the first type of knowledge. Their relationships are modeled after the work of Jackson (1995). Node `R1` – understanding the requirements of the world – is a leaf subskill that is concerned with understanding the *raison d'être* of the program to debug: this includes user requirements, the description of the goals of an exercise, the problem the program is meant to "solve", etc. Node `R2` – understanding the requirements of the notional machine – refers to understanding the constraints in which the program operates. Notional machines are mental models ("machines" in Alan Turing's terminology) that people employ to understand how specific programming languages are executed on specific execution engines (Dickson et al., 2022). These "mental" machines abstract away much of the actual complexity of concrete machines (thus making the CL of understanding them manageable). They may of course be incomplete, erroneous, too coarse to be useful in certain situations or too detailed – hence source of extraneous CL – for others. Consequently, they may, and should, evolve as learners' expertise increases. As notional machines are defined starting from concrete programming languages and the underlying architecture of the machine, `R2` is enabled by `P0`. According to Jackson, the specification of a program – node `R3` – is defined in the intersection of the requirements of the world and the notional machine, Figure 6 provides a visual example of this. This subskill is concerned with understanding how the intended algorithm models the user requirements/goals of the program within the constraints of the machine/execution environment. This is the realm of algorithms and design patterns. A competent application of subskill `R3` entails, for example, being able to explain "in plain English" how a program to debug *should* work. To give a concrete example, this means stepping from a description of a program such as "this program sorts an array of integers in ascending order" (`R1`) to "the function `sortArray` receives the unordered array as input, implements the standard Bubble Sort algorithm and returns the ordered array as output" (`R3`). Note that the latter description may often be given *even if the actual implementation of the algorithm is buggy*, provided that the density of bugs is not so high as to obfuscate the original intent.

3.3.3. *The actual program: T1, T2, T3, T4.*
Li et al. (2019) relate the following subskills to *system knowledge*, i.e., understanding the program to be debugged. This branch of the model is consistent with Lister's three-stage model (Lister, 2021). Nodes `T2` and `T4` represent, respectively, the ability to work at Lister's *tracing* and *post-tracing* levels. The latter

subskill is also part of the *topological knowledge* necessary to orient oneself in a complex system/program (Li et al., 2019). Furthermore, we identify two fundamental prerequisites for tracing: control flow (`T1`) – arguably part of *domain knowledge* together with `P0` (Li et al., 2019) – and data flow (`T3`), which is part of the *functional knowledge* necessary to understand the causal relationships between partitions of the program (Li et al., 2019). The relationships among these nodes are based on how the subskills coordinate in *imperative* programming languages. The analysis of the control flow of a program is a prerequisite to the analysis of its data flow and to tracing it mechanically. Abstract tracing requires the coordination of mechanical tracing and data flow analysis in order to establish abstract properties of a program, such as pre/postconditions and invariants. In fact, for data-driven programming languages, control flow analysis may not be necessary, as the motor of execution is data availability instead of instruction availability[4].

3.3.4. *The debugging process: D1, F1, I1.*
The higher-level nodes – directly connected to the `Debugging` root – represent the macro-level skills that constitute the debugging *process*. Node `D1` – diagnosing the bug – entails being able to "demonstrate" the presence of a bug in a program. This subskill relates to the ability to prove general properties of programs, which also involves being able to reproduce failures, i.e., proving that specific inputs lead to specific final states. The proofs range in complexity, starting from simple observations related to the requirements, passing through assertions and unit tests, and ending in demonstrations of (in)correctness of the type discussed by Hoare (1969). The prerequisites to `D1` are being able to understand the *intended* program and to compare it to the *actual* program to verify discrepancies (failures) and identify their causes (faults/bugs). Node `F1` – applying a fix – entails actually modifying the program to remove the bug. This subskill has much in common with general programming competence. Thus, following Lister's three-stage model, it is enabled by the subskills related to tracing. `F1` is also enabled by `R3`, as the act of "fixing a bug" involves attempting to make the *actual* program converge towards its *intended* version. `D1`, `F1` and their cyclical interplay, are part of the *strategic knowledge* necessary for debugging discussed by Li et al. (2019). Finally, node `I1` – using tools – entails interacting with any programming/debugging instrument such as terminal interfaces, text editors, debuggers, etc. Li et al. categorize this subskill under the *procedural knowledge* necessary for debugging. Note that no subskill in our model *strictly requires* the use of specific tools. This motivates the placement of `I1` as a prerequisite only to the `Debugging` root. However, using tools does indeed contribute to the CL of learning activities. Hence, the role of `I1` in a learning task should not be underestimated.

Nodes `D1`, `F1` and `I1` are all in simultaneous connection, as the coordination of related skills is a prerequisite to the successful debugging of a program. In

[4]For those languages, the relationship between data flow (`T3`), mechanical tracing (`T2`) and abstract tracing (`T4`) would be a linear dependency.

particular: the outputs of D1 and F1 feed into each other as inputs, as programs may need to be retested after a bug fix. As stated above, I1, depending on the specific tools used, is always active and part of the CL of any debugging process involving tools of any kind.

3.3.5. *Applicability to competitive programming settings*
In Olympiad contexts, several subskills have characteristic manifestations. R2 – understanding the requirements of the notional machine – includes machine- and language-level constraints relevant to common contest ecosystems (e.g., C++ and Python): integer overflow and type ranges, floating-point precision, recursion depth, stack vs. heap behavior, and fast I/O. R3 – understanding the specification – entails the careful interpretation of problem statements and the derivation of boundary conditions from input constraints. T4 – abstract tracing – entails the articulation of pre/postconditions for solution components. D1 – diagnosing the bug – encompasses designing minimal counterexamples and adversarial tests to expose deviations from expectations. Finally, I1 – using tools – is often constrained during contests, which shifts emphasis toward lightweight techniques (e.g., localized print-debugging) during the event and toward offline generators, checkers and sanitizers during training. These mappings do not alter the model but clarify how subskills manifest in typical Olympiad workflows.

3.4. *Mapping subskills to competitive programming contest verdicts*

The following examples refer to typical verdicts that may be assigned to programs submitted in competitive programming contexts (Németh & Zsakó, 2018). Common online-judge verdicts – when different from **Accepted (AC)** – may be interpreted as signals of gaps in specific subskills, thus informing targeted remediation and curriculum sequencing:

- **Compilation Error (CE)**: The submission failed to compile (e.g., syntax/semantic errors). Related to P0 (syntax/semantics) and I1 (toolchain familiarity).
- **Runtime Error (RE)**: The program crashed or terminated abnormally (e.g., segmentation fault, division by zero, uncaught exception). Related to T1/T2/T3 together with R2 (e.g., undefined behavior, out-of-bounds access, invalid states).
- **Wrong Answer (WA)**: The program ran but produced incorrect output on at least one test. Related to R3 (specification boundaries and corner cases), T4 (invariants, pre/postconditions), and D1 (counterexample construction and hypothesis testing).
- **Time Limit Exceeded (TLE)**: The program did not finish within the time limit (often due to inefficient algorithms). Related to R3 (problem constraints and required complexity), with diagnosis via D1 using input-size reasoning; often indicates the need for algorithmic redesign beyond F1.
- **Memory Limit Exceeded (MLE)**: The program used more memory than allowed (e.g., oversized data structures, leaks). Related to R2 (memory model and data-

structure footprint), `T3` (data dependencies), and `D1` (space-usage reasoning and simple profiling heuristics).

This alignment helps instructors select which subskills to emphasize when particular verdicts recur in training logs, and it supports designing part-task drills prior to whole-task contest practice.

4. Exercises to Train and Evaluate Subskills

4.1. *Introduction*

This Section illustrates examples of exercises that may support practice and/or be used as candidate assessment tasks for specific subskills of the model. Note that at this point in time these exercises have not been experimentally validated as training/assessment tools, so instructors should use them as sources of inspiration rather than sources of "truth".

Training subskills in relative isolation, before recombining them, is connected to the *isolated elements effect* of CLT (Sweller et al., 2019): its main tenet is that the CL caused by a multifaceted skill is *greater* than the sum of its parts, as the *coordination* of subskills has its own cognitive weight. Strengthening subskills in a relatively isolated way decreases their intrinsic CL, thus leaving room for the CL imposed by learning how to coordinate them. Being able to evaluate learners' competence in the subskills of debugging is also an indirect measurement of the CL that they will cause in a "complete" debugging activity. This may be useful to decide how to structure and scaffold exercise materials.

The learning activities described in the following paragraphs should be structured according to the indications of the 4C/ID (van Merriënboer et al., 2002). Recurrent skills (R) are generally best addressed with small, drill-type exercises built to provide immediate corrective feedback when answered incorrectly. The immediacy of the feedback is necessary to enable learners to keep all the relevant information in WM (the goals of the exercises, the erroneous answers and their motivations, the corrective feedback). This concept is also captured by the *transient memory effect* of CLT (Sweller et al., 2019). These exercises should benefit from the application of the previously discussed *isolated elements effect*. Nonrecurrent skills (NR) are best addressed by learning activities that require the coordination of other subskills. These activities should be carried out after learners receive all the *supportive information* necessary to understand them, which should be studied *before* the learning activity (to avoid WM overload). The activities themselves may be directly carried out by learners – e.g., problem-solving exercises – but they may also be collective activities, discussion with instructors, etc. We refer readers to the concepts of *worked examples*, *completion problems*, *guidance fading* and the *variability effect* discussed by Sweller et al. (2019).

4.1.1. *Case Study:*
For competitive programming (Olympiad) preparation, we recommend part-task practice on recurrent subskills using short (but frequent) drills with immediate corrective feedback, interspersed with varied learning tasks that require coordinating multiple subskills. Metacognition – i.e., reflection about problems and one's processes to address them before, during, and after a problem-solving activity (Yadav et al., 2022) – should be strongly encouraged when practicing learning tasks, as it is the basis for the generalization of skills (van Merriënboer et al., 2002). Instructors may tag exercises with targeted subskills and related online-judge verdicts (e.g., `R3`/`T4` for WA on hidden tests; `R2`/`T3` for RE or MLE; `R3` + `D1` for TLE; `P0` + `I1` for CE). These tags support data-driven remediation and a principled progression from supported drills to time-boxed, whole-task contest practice.

4.1.2. *A Note on Terminology:*
In the following, we use the term "programming instruction" somewhat vaguely to refer, e.g., to both entire lines of code and atomic operations. We do this intentionally, as the definition of what an "instruction" is depends on the notional machine currently employed by a learner, which evolves with experience and may indeed change as the result of insights gathered during a learning activity.

4.2. *Exercises by subskill*

P0) Understanding the programming language (R)

Prerequisite subskills: none.
Exercises related to understanding the *syntax* of a programming language:

- Identify syntax errors in small – e.g., one-line-long – programs. This may be done by selecting from a list of instructions those that cause compile-time or runtime failures and explaining why. See Figure 2-A for an example: this type of exercise may be used to test learners' understanding of the syntax of a programming language even in "unusual" situations, such as that of instruction number 9[5].
- Given a set of requirements and a visual representation of a program built to meet them (e.g., a flowchart), translate the visual representation of the program to executable code. Variant: find discrepancies between the visual representation of a program (which, in this context, represents the *intended* program) and its translation to executable code (the *actual* program).

Exercises related to understanding the *semantics* of a programming language:

- Group single or small sequences of instructions according to some features: e.g., those that print out output on the console, those that alter the "sequential" control flow, etc.

[5]For which even the authors expected a runtime failure that did not, in fact, occur.

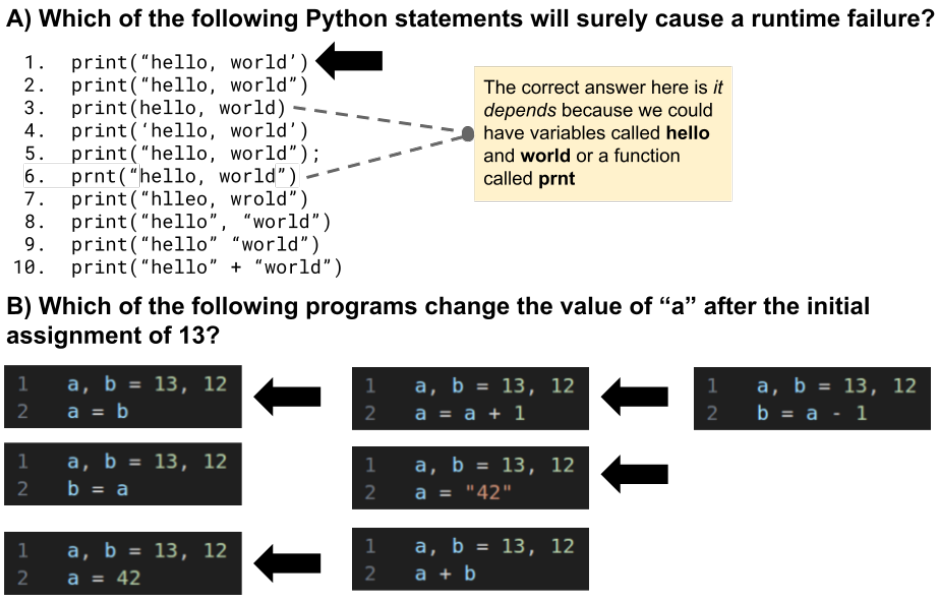

Figure 2. Drill-type exercises targeted at strengthening the understanding of the syntax and pragmatics of a programming language. The arrows mark the correct answers.

- Identify instructions that cause runtime failures due to improper use of operands: e.g., division by 0, or in some programming languages, the "sum" of a string and a number.

Exercises related to understanding the *pragmatics* of a programming language:

- Write *explain-in-plain-English* (where "English" is replaced with the natural language of the learners) descriptions of one-line-long programs. Variant: select from a list statements describing the effects of the execution of one-line-long programs.
- Given a program state – i.e., assignments of values to a few variables – predict how a specific variable's value will change after the execution of a single instruction. See Figure 2-B for an example: the idea of this type of exercise is strengthening the ability to recognize "at a glance" which variables are affected by an instruction.

T1) Control flow (R)

Prerequisite subskills: `P0`.
As the control flow of an execution is often affected by the values that variables take at specific points of the execution flow, the distinction between exercises related to `T1` and `T2` is blurred. We believe, however, that it makes sense to initially focus on control flow in a relatively isolated way, e.g., by analyzing programs that

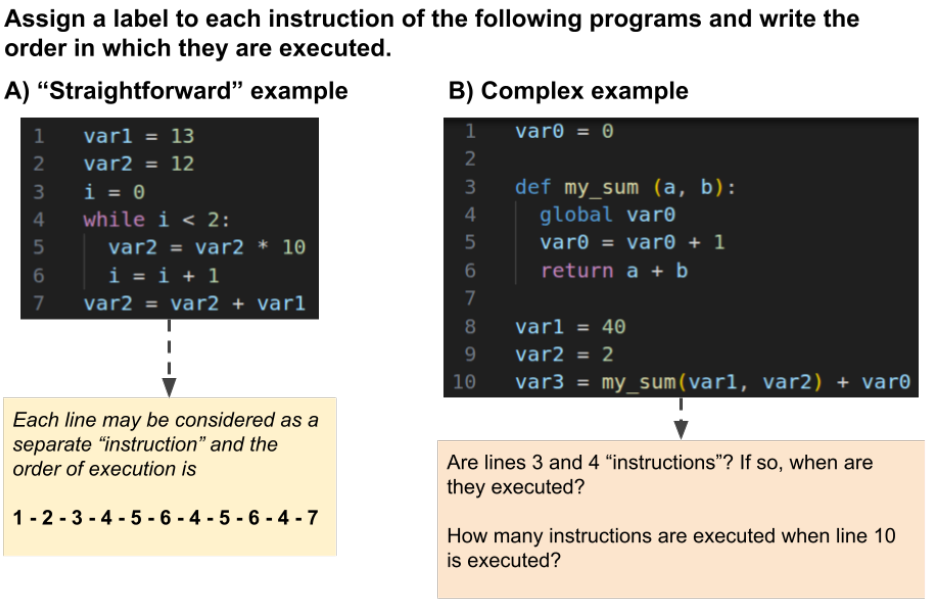

Figure 3. Exercises aimed at strengthening the understanding of the control flow of a programming language.

use constant values only. This is somewhat similar to the approach discussed by Lister (2021), specifically the examples around teaching iteration on arrays.

- Given a simple program, assign a label to each of its instructions and write down the order in which they are executed. Note: what actually constitutes an *instruction* depends on the notional machine employed by the learners at a specific point in time. Depending on their level of expertise and the complexity of the programs to analyze, it may make sense to consider each line of the program as a separate instruction or to break each line down to multiple instructions (e.g., the assignment to a variable of the result of a sum of values). Figure 3 showcases examples of such exercises. Example A is a "straightforward" exercise that may be used to test the understanding of the control flow of a `while` loop. Example B is an admittedly unusual program that however may be used to induce the "conceptual jump" to a more sophisticated notional machine in which "instructions" may no longer be equated to "program lines".
- Draw a control flow graph for a program in which each node represents an instruction, again depending on the notional machine employed at that point in time, and each edge represents a possible sequence of execution.

T2) Mechanical tracing (R)

Prerequisite subskills: `P0`, `T1`.

The exercises for this subskill revolve around *tracing tables* in which each column represents one step of the execution of a program (again, depending on the notional machine), each row represents a variable of that program and each cell represents the value of one specific variable at one specific stage of the execution.

Building tracing tables is a useful exercise to explicitly strengthen and evaluate learners' understanding of how programs are executed step-by-step (Lister, 2021). They are also useful tools to diagnose bugs, as they allow learners to single out the exact moment in which the behavior of the *actual* program begins to deviate from the expectations of the *intended* one.

T3) Data flow (R)

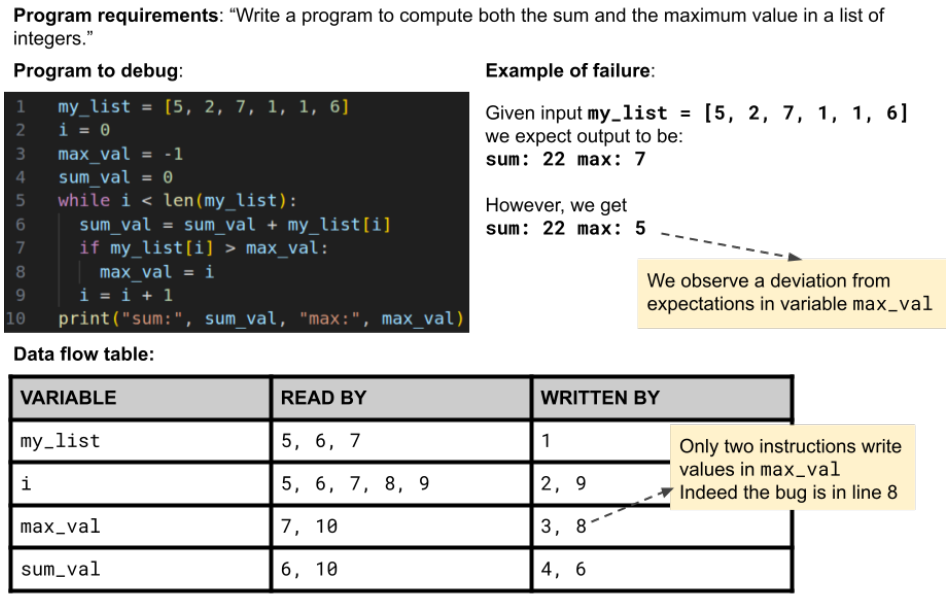

Figure 4. Example of exercise employing data flow analysis as a debugging tool.

Prerequisite subskills: `P0`, `T1`.

- Produce a *data flow table* for a program in which each row represents a variable, one column is used to list all the instructions that *read* that variable and another column is used for all the instructions that *write* it. Data flow tables are useful tools to support program-analysis techniques such as *backward tracing* (Li et al., 2019): after identifying a deviation of the program state from expectations, the table may be used to see which instructions affected the values of particular variables, tracing back to the cause of the bug. Data flow tables are also a

first step in understanding a program as a set of interrelated but also partially independent logical partitions, by making explicit, e.g., that some variables are only affected in writing by a subset of the instructions of the program. See Figure 4 for an example of data flow analysis used in the context of debugging.
- Draw a data flow graph for a program in which each node represents an instruction and each edge represents a read/write dependency.

T4) Abstract tracing (R/NR)

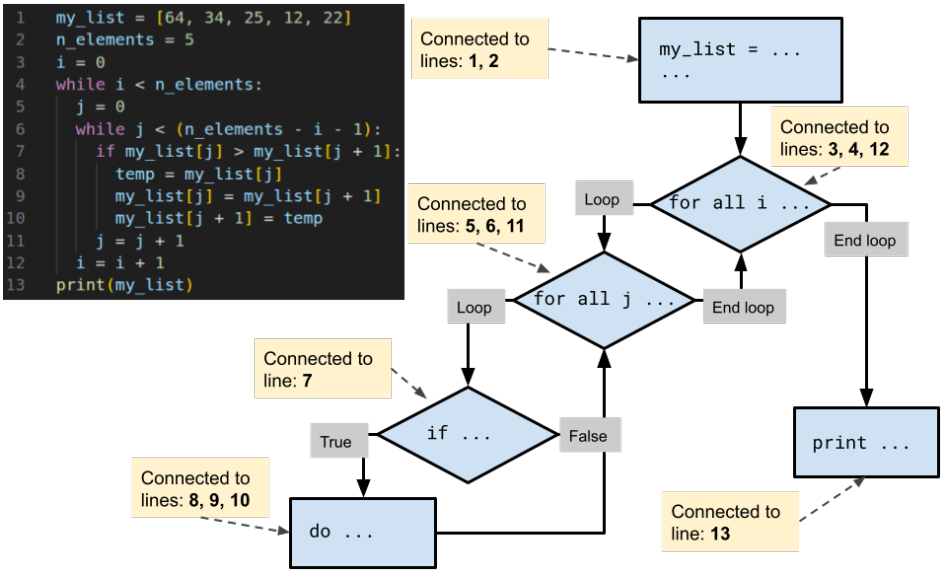

Figure 5. Example of exercise requiring learners to analyze a program as a set of logical partitions.

Prerequisite subskills: `P0`, `T1`, `T2`, `T3`.
This subskill is somewhat hybrid in its recurrent/nonrecurrent nature, as some parts of it, such as establishing pre/postconditions and invariants, are generally straightforward and may be the subject of drill-type exercises. Other parts, such as dividing a program in logical partitions, may, however, be subject to interpretation and opinion, especially for big and complex programs, for which a nonrecurrent approach would probably work best.

- Given a "coarse-grained" flowchart representation of a program (in which each node collates multiple instructions) and an executable implementation of the same, connect each instruction of the latter to a node of the former. This type of exercise may support practice of learners' ability to see a program as a set of interconnected partitions. The example provided in Figure 5 highlights how these

partitions may be composed of logically-related but topologically separated – i.e., non-consecutive – instructions.
- Given a program, divided in a set of interconnected partitions (e.g., procedures, but also logically-grouped sets of instructions), write pre- and postconditions for all of the partitions. Pre/postconditions define *interfaces* and *expectations* for subprograms that may then be verified, e.g., with unit tests.

R1) Understanding the requirements of the "world" (NR)

Prerequisite subskills: none.
This subskill falls partially outside the realm of Informatics, as it is related also to understanding the domain in which a program should operate: for example, calculating the mean of a list of integers requires knowing what a "mean" is.

- Given a machine with which learners are already familiar – e.g., an elevator, a coffee machine, a digital watch – write down and discuss a list of user requirements for that machine.
- Compare the requirements of different "domains" in which programs may operate: e.g., what is the difference between the real-time accuracy requirements of a program that runs a digital alarm clock and those of a program operating in an airplane or nuclear reactor?
- Use the exercises described by Prather et al. (2019): given a description of the requirements of a program and an example input, compute the expected output "by hand", i.e., without executing the program. This type of exercise is useful to verify that learners fully understand the requirements of a program and the domain in which it should operate before debugging it.

R2) Understanding the requirements of the notional machine (NR)

Prerequisite subskills: `P0`.
In general, the use of notional machines as tools to discuss and understand programming systems is a powerful support for learners' metacognition: it makes explicit the fact that people who interact with the "machines" of Informatics use *abstractions* to understand them and that these abstractions are not fixed forever but may change depending on the level of personal understanding and the necessities of the tasks at hand.

- Analyze and discuss notional machines at different levels of detail and complexity for the same programming language. One example is the multiple possible definitions of the word *instruction* in the context of programming.
- Analyze, discuss and compare notional machines for different programming languages and computing agents.

- Discuss why in the Python programming language the instruction `0.1 + 0.1 + 0.1 == 0.3`, when executed, evaluates to `False`.[6] Assuming otherwise could be a plausible cause of a "machine-related" bug in an otherwise logically correct program.

R3) Understanding the specification (NR)

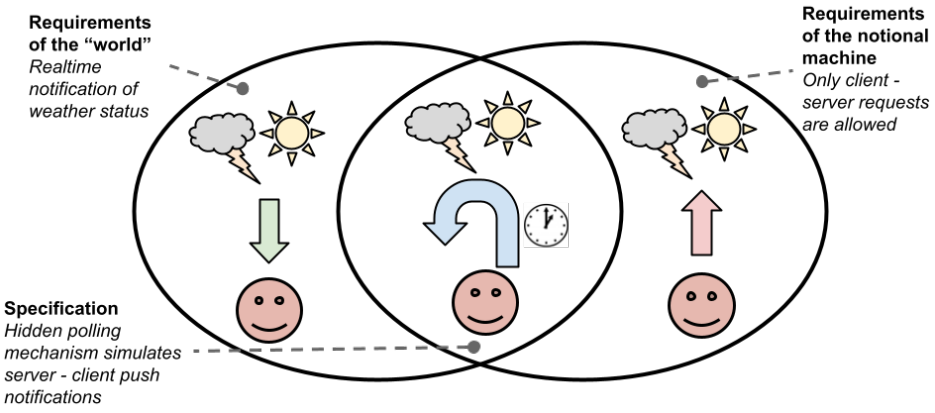

Figure 6. Example of scenario that may be used to conduct a learning activity on program specifications.

Prerequisite subskills: `P0`, `R1`, `R2`.
Understanding specifications and, indeed, *intended* programs, is a subskill that is strongly supported by the *cultural* knowledge of algorithms and design patterns. Being exposed to "canonical" solutions to common problems or to the idiomatic ways in which programming languages implement features is fundamental in becoming able to understand what a program (even a buggy one) is attempting to achieve.

- Given a set of statements describing the "world" requirements of a program and a set of constraints of the "machine", create a map of connections between the two sets and outline an algorithm to meet the requirements within the constraints. Examples may be: building a browser-based push-notification system using only client web-level requests (e.g., by implementing a hidden "polling" mechanism), see Figure 6; using a first-in-first-out queue to manage the floors to be visited by an elevator.

[6] This is due to the "constraints" of how floating-point numbers are implemented in this programming language, see, e.g., https://docs.python.org/3/tutorial/floatingpoint.html

- Given a program and a set of requirements that it is attempting to satisfy, indicate which procedures and partitions of the program relate to specific requirements.

D1) Diagnosing the bug (NR)

Prerequisite subskills: `P0`, `R1`, `R2`, `R3`, `T1`, `T2`, `T3`, `T4`.
From the standpoint of CLT, it makes sense to, at least initially, separate "diagnosing" bugs as a specific learning goal, with dedicated learning tasks. This helps unburdening learners' WMs from the CL related to attempting to fix the programs. It also emphasizes the importance of *demonstrating* – i.e., coming up with proof of – the presence of bugs.

- Use one of the tools discussed thus far (tracing tables, data flow analysis, pre- and postcondition testing) to explicitly and systematically demonstrate the presence of a bug in a program. This requires explicitly setting some expectations on the behavior of the program when executed and providing *evidence* of its deviation from the expectations.
- Use multiple tools to diagnose the same bug, and compare them in terms of efficacy, efficiency, and appropriateness for different types of bugs.
- Given multiple erroneous implementations of a procedure whose pre- and postconditions are explicitly stated, attempt to "capture" all the bugs using unit tests.

F1) Applying a fix (NR)

Prerequisite subskills: `P0`, `R1`, `R2`, `R3`, `T1`, `T2`, `T3`, `T4`.
This subskill largely overlaps with general programming proficiency: after having fully diagnosed a bug, in order to actually fix it, we have to be able to design the correct algorithmic sequence of operations and implement it in an executable programming language. The key point of the exercises related to this subskill is understanding that bugs may be fixed by *localized* interventions on the program, thus avoiding the oft-reported "strategy" of debugging programs by deleting them in one fell swoop and reimplementing them from scratch.

- Given a program with an explicitly indicated bug, solve it by modifying as little of the existing code as possible.
- Do the same exercise as the previous point but as a second step after diagnosing the bug (`D1`).

I1) Using tools (R)

Prerequisite subskills: none.
The exercises related to this subskill are largely dependent on the specific tools to be used in debugging. Our main observation is that learning how to use a tool

should be separated from learning the subskill empowered by that tool. To give an example: learning how to use a tool aimed at building tracing tables should be done *after* having understood how to build tracing tables by hand. Otherwise, the knowledge related to the operation of the tool and that related to building tracing tables risk becoming reciprocal sources of extraneous CL.

A couple of examples of exercises related to tools are:

- Report on a program state at various execution stages, using a debugger to set breakpoints. This should be done after having familiarized with mechanical tracing (`T2`).
- Find when a bug was introduced in a repository using `git bisect`. This requires being able to diagnose the presence of the bug with techniques related to other subskills (`T2`, `T3`, `T4`).

5. Discussion

5.1. *Introduction*

The theoretical nature of the model discussed in this work makes it necessary to systematically discuss what it *shows* (those parts of it that we may deem established) and what it only *suggests* (our hypotheses that need to be validated experimentally). We shall do this separately for the three main contributions that it provides to the state of the art on debugging instruction.

5.2. *Contribution 1 - Debugging Decomposition Model*

The first contribution of this work is a CLT-inspired model of debugging seen as a complex nonrecurrent meta-skill that requires the coordination of multiple subskills. The model is built on three sets of components: 1) debugging subskills; 2) subskill categorization as recurrent/nonrecurrent; 3) dependencies and relationships among subskills. As we detail in Section 3, the first set of components (i.e., the choice of subskills) is firmly grounded in the scientific literature. This is also true for part of the third set of components (dependencies among subskills). Specifically:

- The relationships among `R1`, `R2` and `R3` are based on the work of Jackson (1995).
- The relationship between `T2` and `T4` is based on the work of Lister (2021).
- The interplay among `D1`, `F1` and `I1` is based on works describing the debugging process (see Section 2.4).

Supported by the relevant literature, the elements discussed so far represent what this work can claim as *established* results.

The contributions that these findings provide to the state of the art are: 1) a coherent and cohesive systematization of insights from established literature

sources; 2) an original view of debugging that is compatible with, but different from traditional process descriptions. See Table 2 for a mapping of the model's subskills to steps of the debugging processes discussed in the literature.

What the model presented here only *suggests* is mostly related to the second and third sets of components. We hypothesized the recurrent/nonrecurrent nature of the model's subskills based on reasoning about the consistent/nonconsistent outcomes of the applications of such skills. For example, building a tracing table (`T2`) is always the same activity, regardless of the reason why one is to do it; conversely, building a specification for a program (`R3`) very much depends on what the program aims to do, the related notional machine, etc. We also hypothesized some of the dependencies between subskills: while some of them appear to be straightforward, such as having a basic understanding of the programming language (`P0`) as precursor to control flow analysis (`T1`), others may not be so. This second set of results should thus be experimentally validated in future work.

5.3. *Contribution 2 - Exercises for Specific Subskills*

The second contribution of this work is a catalog of exercises to train and evaluate specific debugging subskills, outlined in Section 4. A *strong hypothesis* of the present work is that these exercises may provide plausible ways to evaluate the subskills to which they are related. Our choice of exercises is based on the, intuitively sound, *expectation* that people who are "good" at a specific subskill should be able to solve them satisfactorily. For example, someone "good" at tracing mechanically should have no problem building a tracing table for a program. Thus, we should be able to correlate, with a certain degree of confidence, performance in the exercises to competence in the related subskills.

A less strong *hypothesis* at this point is that using the exercises we discuss for *training* will be an effective way to develop expertise in the related subskills. To give another example: while we expect someone well versed in the intricacies of "machines" to be able to discuss the differences between the notional machine of Python vs. that of Lisp, it is not automatic that studying those differences will help someone reach that level of expertise. Again, the efficacy of our exercises as tools for training should be experimentally validated by future work.

5.4. *Contribution 3 - Implications for Teaching*

In the following paragraphs, we suggest three implications for teaching derived by our model that may be of immediate use to Informatics instructors. At this stage we may only *hypothesize* the value of the model for this purpose; however, the following discussion may be useful as a source of discussion and to build foundations for future work.

5.4.1. *Analyze Existing Learning Materials*

By breaking down the general skill of debugging into constituent subskills, our model enables instructors to analyze existing learning tasks and to estimate their

	Li et al. (2019)	Carver et al. (1987)	Böttcher et al. (2016)	Spinellis et al. (2018)	Michaeli et al. (2019)
P0	Construct the problem space				
T1	Construct the problem space		Find a location to subdivide the system		... find relevant lines of code
T2	Construct the problem space, Diagnose the fault	"Where might the bug be?", Read every command	Find a location to subdivide the system	Find new configuration subset that still yields the failure	... find relevant lines of code
T3	Construct the problem space, Diagnose the fault	"Where might the bug be?", Use the information to find the bug	Find a location to subdivide the system	Find new configuration subset that still yields the failure	... find relevant lines of code
T4	Construct the problem space, Diagnose the fault	"Does the program have subpro-grams?", "Where might the bug be?", Use the information to find the bug	Find a location to subdivide the system	Find new configuration subset that still yields the failure	... find relevant lines of code
R1	Construct the problem space		Formulate an expectation on program state at that location	Form new hypothesis regarding failure's cause, Refine hypothesis	Read and understand the error message
R2	Construct the problem space		Formulate an expectation on program state at that location	Form new hypothesis regarding failure's cause, Refine hypothesis	Read and understand the error message
R3	Construct the problem space, Identify fault symptoms	"What is the problem?"	Formulate an expectation on program state at that location	Form new hypothesis regarding failure's cause, Refine hypothesis	Read and understand the error message, Modify your assumptions or make a new one
D1	Identify fault symptoms, Generate and verify solutions	"What is the problem?", Re-test the program	Use symbolic debugger to check the expectation	Reproduce failure, Test prediction through experiments	"Do expected and actual behavior match?", Determine the error...
F1	Generate and verify solutions	"What should the fix be?", Make the fix	Fix bug and write test case		Adjust your program
I1	Generate and verify solutions		Use symbolic debugger to check the expectation		

Table 2
Possible mapping of the subskills in Figure 1 to components of debugging process models discussed in the literature.

CL. A list of subskills needed to solve a learning task may be determined by starting from the node related to its main learning goal and then traversing all vertical dependencies down to leaf nodes. Each traversed node is a subskill that is very likely activated by the learning task. Instructors may estimate the task-related CL in two ways. First, they may assess their learners' *mastery* of each subskill involved in the task, e.g., by using exercises such as those presented in Section 4. According to CLT, intrinsic CL is inversely proportional to competency. This assessment could be thus used to identify the subskills that are more likely to cause high CL. A similar analysis may be done on how the learning task's materials *scaffold* each subskill. For example, completing a tracing whose rows' and columns' headers are already filled out will likely be less burdensome for novices' cognitive load when compared to a "blank slate" to fill from scratch.

Example of task analysis: consider a learning task that requires the learners to diagnose a bug in an implementation of a classic sorting algorithm such as Bubble Sort. The main learning goal of this task is tied to subskill `D1`. The full list of precursors to `D1` is: `P0`, `R1`, `R2`, `R3`, `T1`, `T2`, `T3`, `T4`. Ideally, an instructor will have an approximate estimate of the learners' mastery of each of these subskills, which may be obtained through periodic quizzes, etc. The learners may, for example, be familiar with mechanical tracing (`T2`), but have only recently been introduced to abstract tracing (`T4`). In this scenario, the learning task may induce different CLs, based on *how* it requires the bug to be diagnosed. If the task provided some example arrays to feed into the buggy algorithm, and asked learners to build tracing tables for each input, then the focus would be more on `T2`. The intrinsic CL related to `T2` would be relatively low, due to the learners' familiarity with the subskill. Conversely, if the request was to diagnose the bug completely from scratch, then the focus would be more on `T4`, as learners would need to at least partially identify a class of inputs that always lead to a failure. The intrinsic CL related to `T4` would be relatively high, and this would affect the overall CL of the task, making it more burdensome. Another important factor in the overall CL of the learning task would be tied to previous knowledge of the implemented algorithm, in this case Bubble Sort. If the learners were familiar with the algorithm, the relative intrinsic CL of subskill `R3` would be much reduced.

5.4.2. *Structure Learning Curricula*

The model's vertical dependencies may be used to guide the design of longitudinal sequences of learning tasks. Starting from the leaf nodes, instructors may design learning trajectories through the subskills: after introducing and focusing on some concepts related to one node, learners are ready to tackle them in the context of a subsequent node enabled by the first one, and so on. The best trajectories through the model's nodes would be spiral-shaped: subskills should not be learned and left behind but continuously revisited, consolidated and expanded upon. This allows instructors to introduce additional subskills and learning tasks within bounded time intervals. Moreover, the repeated practice sessions on foundational subskills

progressively reduce their related intrinsic CL. This allows the ingestion of new concepts and subskills while keeping the overall CL consistent.

Example of spiral-shaped learning trajectory through the model: Lister (2021) provides a detailed example of how to teach iterative processes on arrays according to his three-stage model. The sequence of activities he discusses is compatible with a trajectory on our model. First of all, he states that before tackling arrays, his learners are familiarized with sequences of instructions and assignments. As arrays do not introduce special control flow features, these foundations cover subskill `T1`. The actual teaching example starts by showcasing how arrays are initialized and how values are assigned to specific indexes. This relates to subskills `P0` and `T1`, and is already an example of a new "round" of the spiral. Lister then suggests focusing on examples involving interactions with arrays through *constant* subscripts. Note that this puts the focus on mechanical tracing (`T2`). The next step introduces *variable* subscripts, which require a measure of data flow awareness (`T3`). Only then Lister introduces iteration constructs. This step makes the spiral trajectory go full circle again, as the introduction of iteration requires that learners understand the related programming constructs (`P0`), control flow (`T1`), data flow (`T3`) and tracing mechanics (`T2`, `T4`).

5.4.3. *Design New Learning Materials*

In delineating a learning curriculum for debugging, instructors may need to design learning tasks aimed at strengthening *specific* subskills. Our model clarifies what the prerequisites of each debugging subskill are, i.e., the elements that come into play in related learning tasks. Furthermore, our classification of subskills as *recurrent* and *nonrecurrent*, along with the indications of the 4C/ID, suggests the best *type* of learning task to practice each subskill. The "actual program" branch of our model would be better served by quick exercises designed to give immediate corrective feedback. Conversely, the "intended program" branch of the model consists of nonrecurrent subskills only: these are better strengthened through learning tasks and examples with focus on *variety*. The exercises we detail in Section 4 may be used as a source of inspiration and foundation for the design of new learning activities.

6. Conclusion

The model presented in this work addresses the oft-reported difficulties that arise in learning and teaching debugging. It is motivated by the demonstrated benefits of approaching debugging in a systematic fashion. The main conceptual contribution of this work is a description of the complex and multifaceted meta-skill of debugging seen through the lens of Cognitive Load Theory (CLT): our analysis is firmly grounded in this established theoretical and experimental field and it systematizes various scientific sources in a coherent model that describes a dependency structure of debugging subskills. Our approach offers an original view of

debugging that differs from and complements the process models – i.e., descriptions of debugging as a sequence of actions – established in the literature. Our second contribution is in our extended examples of exercises to support practice of and potentially assess each of the subskills of the model. Furthermore, the model may be used as a tool to analyze existing debugging-related educational activities from the point of view of CLT (e.g., to list all the subskills addressed by an exercise), to plan learning trajectories following the vertical dependencies among the subskills and to design new learning activities. These contributions apply to Informatics Education in general and to higher-secondary/university level CS1 courses in particular: these are the contexts in which learners often first face the complexities of debugging and thus need more educational support.

As a case study, we discussed the potential application of the model to competitive programming preparation contexts. In such settings, the model offers a principled way to distribute cognitive load across a curriculum: build fluency in recurrent subskills through drills with immediate feedback, then integrate them in varied learning tasks that foreground nonrecurrent subskills under contest constraints. The mapping from subskills to online-judge verdicts supports data-driven remediation and the analysis of training logs (e.g., recurring wrong answer/runtime error/time limit exceeded patterns).

Future work: as stated throughout, this work is at the present stage mainly theoretical and needs experimental validation by future studies. The effectiveness of the exercises discussed in Section 4 as training/assessment tools could be studied at first with classroom pilots and then with pre-test/post-test studies. The assessments may be further verified by correlating them with the types of errors made by learners in programming/debugging exercises and with qualitative results gathered from think-aloud debugging sessions. The effectiveness of the model as instructional scaffolding in general may be assessed with pre-test/post-test comparisons of learners' groups receiving "standard" debugging instruction vs. interventions shaped by the model. The dependencies of the model may be verified following the approach of the BRACElet group (Lister, 2021), whose results show strong evidence for the nature of tracing as a prerequisite skill to programming.

Previous dissemination

This work extends an article that originally appeared in the proceedings of the 18th International Conference on Informatics in Schools (Pozzan et al., 2026). The current version expands and details all the content of the conference article, with the full addition of Section 4, most of Section 5, and the case study about competitive programming contexts.

Funding

This research received no external funding.

Competing interests

The authors declare no competing interests.

Ethics approval and consent to participate
Not applicable. This article is a conceptual/theoretical study and does not report primary research involving human participants or identifiable personal data.

Data/materials availability
No empirical dataset was generated or analyzed for this article. The conceptual model and exercise examples are included in the manuscript.

Generative AI disclosure
The authors did not use generative AI or AI-assisted tools in the preparation of this manuscript.

Author contributions
Gabriele Pozzan and Tullio Vardanega conceived the initial version of the model presented in the paper. Gabriele Pozzan and Andreas Bollin developed it to its final state. Tullio Vardanega supervised the work and verified the developed model. All authors contributed to writing the final manuscript.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin*, *37*(3), 84–88. https://doi.org/10.1145/1151954.1067472

Böttcher, A., Thurner, V., Schlierkamp, K., & Zehetmeier, D. (2016). Debugging students' debugging process. *2016 IEEE Frontiers in Education Conference (FIE)*, 1–7. https://doi.org/10.1109/FIE.2016.7757447

Carver, M. S., & Risinger, S. C. (1987). Improving children's debugging skills. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 147–171). Ablex Publishing Corp. https://doi.org/10.5555/54968.54978

Dickson, P. E., Richards, T., & Becker, B. A. (2022). Experiences implementing and utilizing a notional machine in the classroom. *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, 850–856. https://doi.org/10.1145/3478431.3499320

European Commission/EACEA/Eurydice. (2022). Informatics education at school in europe. *Publications Office of the European Union*. https://doi.org/10.2797/268406

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Gauss, E. J. (1982). The "wolf fence" algorithm for debugging. *Communications of the ACM*, *25*(11), 780. https://doi.org/10.1145/358690.358695

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576–580. https://doi.org/10.1145/363235.363259

IEEE Computer Society. (2010). IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061

International Olympiad in Informatics Scientific Committee. (2025). IOI Syllabus [Accessed: 2026-04-14]. https://ioinformatics.org/page/syllabus/12

Jackson, M. (1995). The world and the machine. *Proceedings of the 17th International Conference on Software Engineering*, 283–292. https://doi.org/10.1145/225014.225041

K-12 Computer Science Framework Steering Committee. (2016). *K-12 computer science framework*. ACM. https://doi.org/10.1145/3079760

Laaksonen, A. (2024). *Guide to competitive programming: Learning and improving algorithms through contests*. Springer, Cham. https://doi.org/10.1007/978-3-031-61794-2

Li, C., Chan, E., Denny, P., Luxton-Reilly, A., & Tempero, E. (2019). Towards a framework for teaching debugging. *Proceedings of the Twenty-First Australasian Computing Education Conference*, 79–86. https://doi.org/10.1145/3286960.3286970

Lister, R. (2021). On the cognitive development of the novice programmer: And the development of a computing education researcher. *Proceedings of the 9th Computer Science Education Research Conference*. https://doi.org/10.1145/3442481.3442498

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

Michaeli, T., & Romeike, R. (2019). Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*. https://doi.org/10.1145/3361721.3361724

Németh, Á. E., & Zsakó, L. (2018). Grading systems for algorithmic contests. *Olympiads in Informatics*, *12*, 159–166. https://doi.org/10.15388/ioi.2018.13

Pozzan, G., Bollin, A., & Vardanega, T. (2026). A teaching and learning oriented decomposition of debugging subskills informed by cognitive load theory. In J. Staub & A. Singla (Eds.), *Informatics in schools. fostering problem-solving, creativity, and critical thinking through computer science education* (pp. 153–166). Springer Nature Switzerland. https://doi.org/10.1007/978-3-032-01222-7_12

Prather, J., Pettit, R., Becker, B. A., Denny, P., Loksa, D., Peters, A., Albrecht, Z., & Masci, K. (2019). First things first: Providing metacognitive scaffolding for interpreting problem prompts. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 531–537. https://doi.org/10.1145/3287324.3287374

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2). https://doi.org/10.1145/2483710.2483713

Spinellis, D. (2018). Modern debugging: The art of finding a needle in a haystack. *Communications of the ACM*, *61*(11), 124–134. https://doi.org/10.1145/3186278

Sun, C., Yang, S., & Becker, B. (2024). Debugging in computational thinking: A meta-analysis on the effects of interventions on debugging skills. *Journal of Educational Computing Research*, *62*(4), 1087–1121. https://doi.org/10.1177/07356331241227793

Sweller, J., van Merriënboer, J. J., & Paas, F. (2019). Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review*, *31*, 261–292. https://doi.org/10.1007/s10648-019-09465-5

van Merriënboer, J. J., Clark, R. E., & De Croock, M. B. (2002). Blueprints for complex learning: The 4C/ID-Model. *Educational Technology Research and Development*, *50*(2), 39–61. https://doi.org/10.1007/BF02504993

Yadav, A., Ocak, C., & Oliver, A. (2022). Computational thinking and metacognition. *TechTrends*, *66*(3), 405–411. https://doi.org/10.1007/s11528-022-00695-z

Yang, S., Baird, M., O'Rourke, E., Brennan, K., & Schneider, B. (2024). Decoding debugging instruction: A systematic literature review of debugging interventions. *ACM Transactions on Computing Education*, *24*(4). https://doi.org/10.1145/3690652

Zemanek, H. (1966). Semiotics and programming languages. *Communications of the ACM*, *9*(3), 139–143. https://doi.org/10.1145/365230.365249