Rules of Program Behavior

RODRIGO DURAN, Federal Institute of Mato Grosso do Sul, Brazil and Aalto University, Finland
JUHA SORVA and OTTO SEPPÄLÄ, Department of Computer Science, Aalto University, Finland

We propose a framework for identifying, organizing, and communicating learning objectives that involve program semantics. In this framework, detailed learning objectives are written down as *rules of program behavior* (RPBs). RPBs are teacher-facing statements that describe what needs to be learned about the behavior of a specific sort of programs. Different programming languages, student cohorts, and contexts call for different RPBs. Instructional designers may define progressions of RPB rulesets for different stages of a programming course or curriculum; we identify evaluation criteria for RPBs and discuss tradeoffs in RPB design. As a proof-of-concept example, we present a progression of rulesets designed for teaching beginners how expressions, variables, and functions work in Python. We submit that the RPB framework is valuable to practitioners and researchers as a tool for design and communication. Within computing education research, the framework can inform, among other things, the ongoing exploration of "notional machines" and the design of assessments and visualizations. The theoretical work that we report here lays a foundation for future empirical research that compares the effectiveness of RPB rulesets as well as different methods for teaching a particular ruleset.

CCS Concepts: • **Social and professional topics** → **Computer science education**; *Model curricula*; Student assessment;

Additional Key Words and Phrases: Programming education, introductory programming, rules of program behavior, model of program behavior, learning objectives, instructional design, notional machines, semantics

ACM Reference format:
Rodrigo Duran, Juha Sorva, and Otto Seppälä. 2021. Rules of Program Behavior. *ACM Trans. Comput. Educ.* 21, 4, Article 33 (November 2021), 37 pages.
https://doi.org/10.1145/3469128

PRELUDE

Four people, four questions:

- **Amal:** What exactly should I teach about program execution to the beginners in my class?
- **Bao:** I'd like to share my course design with other programming teachers. What information would be useful for them to know?
- **Camille:** For my research, I'm collecting assessment results from Amal and Bao's courses. The courses seem to cover the same topics, but do they really and what are the differences?

This work was partially funded by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), grant number 201365/2015-4.
Authors' addresses: R. Duran, Federal Institute of Mato Grosso do Sul, Brazil, and Aalto University, Finland; emails: rodrigo.duran@ifms.edu.br, rodrigo.duran@aalto.fi; J. Sorva and O. Seppälä, Department of Computer Science, Aalto University, Finland; emails: {juha.sorva, otto.seppala}@aalto.fi.

1946-6226/2021/11-ART33 $15.00
https://doi.org/10.1145/3469128

- **Dana:** There's this tool that illustrates program runs, which might help my students, but is it a good fit for what I'm trying to teach?

The people are imaginary but the questions are real. In this article, we chart some progress toward answering them.

1 INTRODUCTION

A programming environment is a user interface that enables its users to specify a system's future behavior. To learn to program, the user must learn the capabilities and limitations of that interface. They must learn to reason about what the system does as it runs a program and to predict the effects of each instruction on the system. They need a model of program behavior [64, 132].

The solution is not that all or most learners study the formal semantics used by programming-language researchers or the innumerable details of hardware. Rather, learners need a model of program behavior that is based on concepts within the learners' grasp and that explains the kinds of programs that the learners read and write.

Consider, for instance, this tiny Python program:

```
for number in range(10):
    print(number)
```

Code just like that shows up in many introductory courses. However, different courses discuss it in different terms—and indeed aim for different readings of such examples. One course might teach that `number in range(X)` is a way of covering the integers smaller than X and leave it at that; another course might emphasize that `for number in` is an alternative way of defining a local variable; a third would expect students to learn that `range(10)` is a function-calling expression; and a fourth would elaborate on the fact that any suitable object reference may follow `in`. Saying that a course "teaches students `for` loops in Python" might mean a range of different things.

Given the need and demand for computing education worldwide, practitioners' efforts to share ideas with their colleagues are highly valuable, as is research that evaluates different pedagogical approaches. In both activities, context is paramount: There is no universal best practice [22]. Whether a pedagogy works depends on specific goals and specific circumstances. It is thus unfortunate that we researchers and educators have so often neglected to give much attention to the specific models of program behavior that are being taught. Learning objectives for programming courses are commonly documented only in terms of high-level concepts (e.g., iteration) or syntactical constructs (e.g., `for` loops). Similarly, many research articles describe programming courses by merely listing which high-level topics were covered in each week. Curricula, course syllabi, and even lesson plans are typically vague about how students should reason about programming constructs.

In this article, we argue that it is a good idea to set down, in some detail, what learners are expected to learn about program behavior: The rules that govern what the programmable system does with the programmer's instructions. Doing so has the potential to benefit instructional design, teacher collaboration, and **computing education research (CER)**. It will not fully answer Amal, Bao, Camille, and Dana's complex questions, but it should take us a step closer to having better answers.

2 ARTICLE STRUCTURE AND GOALS

This is not an empirical paper as such. Instead, we formulate a framework that is motivated and informed by prior empirical research and cognitive theories of learning and that can instruct and inspire further research.

We contribute to the literature in four ways.

First, we propose a framework for identifying, organizing, and communicating learning objectives that involve program semantics. These learning objectives are written down as **rules of**

When it runs a turtle-movement command, the computer makes the turtle and its pen move instantly onscreen.

When the computer evaluates an expression, it essentially substitutes the expression in the running code.

An expression is a piece of code that the computer can evaluate to produce a value.

Applying a function `f` means replacing the expression with `f`'s body, with all occurrences of `f`'s parameters therein replaced by the argument values. (Adapted from [44].)

Running programs live in memory, which is divided between a call stack and a heap. (Quoted from [149].)

A Java variable is much like Python variables (which the learners know) except that it has a fixed data type.

In Scratch, when the green flag is clicked, *all* scripts starting with the green-flag block run.

An object responds to a message by following a specific algorithm. It may modify state, send messages to other objects, etc. Finally, it responds with a return message, at which point the calling object resumes its task.

You can assign a list to a variable; this 'names' the list. The computer uses such names to identify which list you mean.

A list-creating expression evaluates to a new *reference*, which is essentially a number that the computer uses behind the scenes to note where in memory the list is. The programmer can store the reference in a variable.

A method is a function on an object. Calling a method requires a reference to the target object, a method name, and arguments. The computer runs the method like any other function; the target object is just another argument.

To evaluate a binary operator like ==, the computer pops two values off the accumulator stack, computes the result, and pushes it onto the stack. (Adapted from [97].)

A class is a *template* for creating a new objects in memory. *Objects are distinct from it; a class is not a collection of, or container for, objects (as learners may assume).

Fig. 1. Statements that could be used as RPBs in different contexts in combination with others not shown here. (N.B. We are not claiming that these are excellent RPBs.)

program behavior (RPBs): teacher-facing, context-sensitive statements about the execution of a specific sort of programs. Related rules are grouped into *rulesets*. Rulesets can be taught in *progressions*, with a variety of different teaching methods possible for any given ruleset. Section 3 introduces the RPB framework in more detail before we get to related work in Section 4, where we discuss how our framework derives from—and provides structure to—earlier research in computing education and the work on "notional machines" in particular.

Second, Section 5 presents a proof-of-concept progression of RPB rulesets.

Third, we contribute a discussion on the design of RPB rulesets and progressions. Section 6 examines how the choice of RPBs is affected not only by the programming language but also the overall goals, audience, and other contextual factors. We highlight the importance of pedagogical content knowledge in RPB design and point up a number of criteria for evaluating designs.

Fourth and finally, in Section 7, we envision how the RPB framework may help computing educators to improve and share their practices as well as helping computing education researchers to ask more incisive questions and design better studies, assessments, and software tools.

Our main focus is introductory-level programming. However, we believe that the RPB concept has potential in other contexts as well, such as intermediate or advanced courses where learners switch between programming languages or paradigms, and we will comment on this topic intermittently.

3 DEFINITIONS

3.1 Rules of Program Behavior

RPBs are written statements about how computer programs of a particular kind behave when executed. Figure 1 shows an assortment of examples.

RPBs specify *learning objectives* for programming education. In particular, they break down the generic objective that students should learn what programs do, which is required for reading and

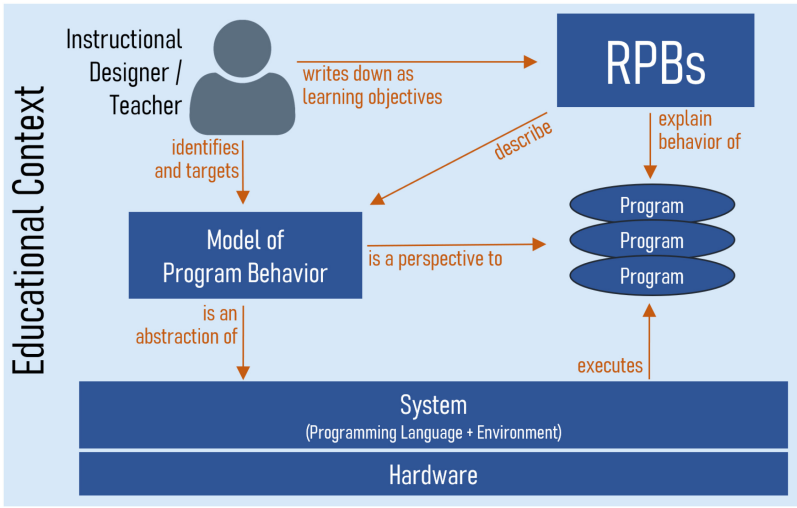

Fig. 2. Rules of program behavior are learning objectives designed for a context. They describe, in a teacher-facing way, selected aspects of program behavior that should be taught.

writing programs. RPBs elaborate on descriptions such as "The learner should know assignment statements" by giving semantics to programming-language constructs. In other words, the teacher identifies the specific content that will help learners reason about what happens at runtime. That content is the target *model of program behavior* and can be written down as RPBs.

RPBs are *teacher-facing*. They should be understandable by teachers and assist in instructional design. RPBs may also help teachers and researchers document and share their designs (Figure 2).

RPBs serve a *pedagogical purpose*. They apply to a *specific sort of programs*, such as those written in a certain programming language or subset of a language. They describe program behavior at a level of abstraction that helps state what the learners should learn; that description may be more or less informal and more or less detailed, as decided by the RPBs' author.

RPBs are *sensitive to context*. Different programming languages and environments, different programs, different cohorts of learners, and different curricular goals call for different RPBs. A set of RPBs designed for a context may be useful in another sufficiently similar context. RPBs may target a specific point in a curriculum (e.g., the end of a course or "after Week 2") at which learners should reach a certain understanding. RPBs may be organized into *progressions* that trace a planned trajectory of detailed learning objectives for different stages of a course or curriculum.

RPBs do not reflect only the technical aspects of the system; They also embody their author's *pedagogical content knowledge* [119, 126, 151]. For instance, RPBs may list "negative goals": What students need to learn the system does *not* do, even though a student might assume otherwise.

3.2 Progressions

Introductory-level programming concepts are interdependent, perhaps even more so than in other disciplines [93, 117]. It is difficult or impossible to write individual RPBs that are independent of other concepts and rules. Since we would like students to make progress toward a consistent, widely applicable understanding of a programming language, it often makes sense to design *rule-sets* rather than isolated rules. A ruleset is a collection of RPBs that have been designed to work together and that describe a perspective to program behavior.

§1 The program text you write dictates the computer's behavior when it runs your code, but the behavior involves stages and concepts that aren't spelled out in code.
§2 An *expression* is a piece of code that the computer can *evaluate* to produce a *value*.
§3 A value is a granule of data that cannot be further simplified by evaluation.
§4 You can write expressions into code. The computer evaluates them when it runs that code.
§5 The REPL does evaluation for you: type in an expression to get (a description of) its value.
§6 Evaluation happens in a section of computer memory, an *'evaluation area,'* where the computer stores intermediate results during evaluation.
§7 *Arithmetic expressions* are a kind of expression that is composed of smaller expressions. The computer evaluates such an expression by doing the arithmetic.
§8 A *literal* is a kind of expression where a value is written 'as is' into code. Its evaluation is straightforward.
§8.1 Nevertheless, a literal too *is* evaluated, e.g., to get the integer represented by a sequence of digits.

Example code:

```
>>> 1 + 1
2
>>> 2 * (10 + 5)
30
```

```
>>> 100
100
>>> 1*2*3*4*5*6*1*2*3*4*5*6*1*2*3*4*5*6
373248000
```

Fig. 3. Example ruleset #1: Expressions, values, and evaluation in python. The example code in the Python REPL (Read-Eval-Print Loop; interactive interpreter) illustrates the programs that the ruleset applies to.

Figure 3 shows a ruleset that we include here as an illustration, not as an exemplar of high-quality RPB design. (Quality depends on context and involves tradeoffs, which we will discuss in Section 6.) For ease of reference, we have chosen to express Figure 3 and other rulesets in this article as lists of numbered rules, but we do not mean to imply that rulesets *must* be enumerated so.

An RPB progression is a sequence of increasingly powerful rulesets that plot a learning trajectory. Each ruleset in a progression builds on the previous one but has more rules and/or its rules are more detailed or otherwise different. It is more powerful than the progression's earlier rulesets in that it applies to a greater variety of programs, is more accurate, or otherwise represents a more ambitious learning goal. Not all RPB rulesets need to be part of a progression, but we will argue in this article that designing progressions is often a good idea.

An RPB progression may be *local* to a programming language and course or it may be *scopious*, covering multiple courses or languages. A local progression helps educators plan and communicate the pedagogy of a single course or a part thereof. Depending on needs, a local progression might have just a few rulesets that describe major milestones or it might detail each evolutionary step at a fine grain. The ruleset of Figure 3 is part of a local progression whose design we will introduce later in Section 5. (The reader may wish to peek ahead at Figures 5–8.) A scopious progression has a coarser grain and describes the learning objectives at the ends of courses or other key points within a broader curriculum. It highlights what students know coming in to a course and what they need to learn during each course.

A local progression is often *monolingual*: It blazes the trail that students take as they learn about one language and its rules. Even for a single language, many paths are possible, which is now even more apparent given the popularity of so-called multi-paradigm languages such as JavaScript and Python. For example, a monolingual RPB progression might take learners from imperative scripts to "everything is an object," or from immutable to mutable state, or even from dynamic to static typing, all within a single language (cf. Reference [83]). A scopious progression is often *multilingual*: Learners use different languages in different courses, so all the rulesets do not apply to the same language.

3.3 Clarifications of the RPB Concept

RPBs are conceptually distinct from any student-facing materials that are used to teach the RPBs; see Figure 4. (Some rulesets may be accessible to some students in their teacher-facing written

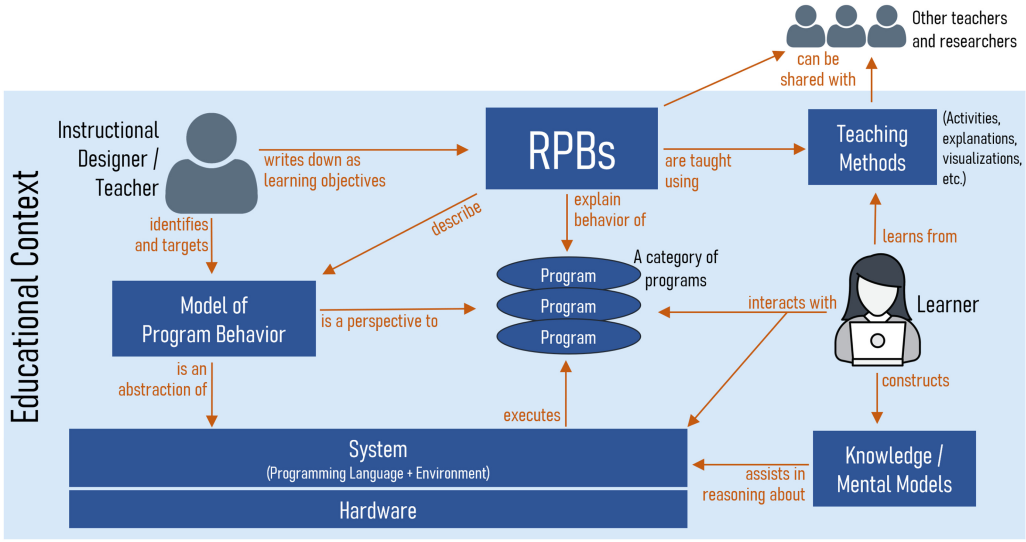

Fig. 4. RPBs are a basis for didactics. However, a set of RPBs is distinct from the various methods for teaching it. (This diagram extends Figure 2, whose contents appear as the left-hand side of this diagram.)

form, but that is incidental to the RPB framework.) Pedagogy that is based on a ruleset may employ different kinds of activities, explanations, or visualizations. Different teachers may target the same ruleset but use different methods that match their contextual needs or personal preferences.

Since an RPB ruleset is tied to a context and a purpose, it does not need to describe a universal model of computation. Neither is a ruleset a generic model of the underlying hardware or operating system. Powerful abstractions separate modern programmers from a myriad of implementation details, and the RPB framework relies on that. A ruleset captures selected aspects of the programming language and its associated tools and runtime environment, at a level of abstraction that is helpful for instructional design. It records only those high-level properties of the system that matter for the sort of programming that the learners will engage in.

The concept of RPBs is similar to programming-language semantics, which define formal rules that describe what programs do and mean. An RPB ruleset can be thought of as a teaching-oriented, more or less informal relative of operational semantics that applies to a certain context.

Many formal semantics seek to cover an entire language as completely as is feasible. In contrast, an RPB ruleset typically focuses on a narrow set of aspects of a language and an environment. A ruleset will often be quite incomplete—i.e., it will not explain nearly all programs in the language. Like a semantics, RPBs can be unsound in some respects; it may cut corners for educational reasons.

As noted, RPBs are not independent of programming language. However, they are not necessarily tied to only one language: Just as different programming languages can have the same semantics, an RPB ruleset designed for one language may work for a similar language.

Program behavior is just one aspect of programming languages that influences learning. Other aspects include syntax, code organization, and input modality (e.g., text or blocks) [83]. The RPB framework focuses on behavior while acknowledging that other aspects may impinge on understandings of behavior and, consequently, behavior-related learning goals and teaching methods. For instance, while syntax does not uniquely determine program behavior, it affects people's learning about behavior. The purpose of RPBs is to explain behavior, not syntax; it is up to the RPB author to decide whether it helps to refer to syntactic elements when phrasing RPBs.

4 RELATED WORK IN CER

In this section, we look at related work and consider its ties to the RPB framework presented above. The first (Section 4.1) discusses beginners' difficulties with tracing program behavior. The second (4.2) overviews theories of conceptual change and beginner programmers' misconceptions. The third (4.3) presents advice from research on causal systems. The next two subsections compare RPBs to related concepts in the CER literature: notional machines (4.4) and progressions of sublanguages (4.5). Finally, we comment on how teaching methods link to the RPB concept (4.6).

4.1 Reasoning about Program Behavior

Tracing a program means reasoning systematically about a program's behavior and predicting what the program will do. Tracing is a component skill that is required for many programming activities; learning to trace contributes to learning other programming skills. A student who cannot trace reliably will have a hard time debugging programs, learning from examples, and writing programs of their own, except by copy-paste and trial-and-error.

Beginner programmers frequently fail at tracing, either neglecting it entirely or being unsuccessful at the attempt [24, 25, 52, 89, 105]. One of the underlying reasons is that many beginners struggle to understand the relationship between a program and the runtime behavior that it specifies; learners (and their teachers) may focus excessively on the visible program and neglect the intangible concepts that give that program meaning [5, 41, 132].

Before they have learned of and sufficiently practiced better alternatives, many beginners intuitively fall back on behaviors that work in natural-language conversation or math class but fail when applied to program code (see, e.g., References [36, 88, 94, 102]). The division of labor between the programmer and the system that interprets and runs the programs is often not obvious: Learners frequently overestimate the capabilities of the system and underestimate the precision required in instructing it. Left to their own devices, beginners invent ad hoc tracing strategies that, with luck, may work but that are often unsound or inefficient [20, 24, 52, 88, 145].

The lack of viable knowledge of program behavior is an obstacle to learning and a source of frustration and unproductive failure. There is also some evidence linking conceptual difficulties to lower programming self-efficacy [79] and hard-to-fix bugs [42].

It thus makes sense to acknowledge a model of program behavior as a learning objective. Since there are many possible models with different characteristics, teachers need to make choices. However, although programming curricula and course designs often list language constructs and other content, they usually do not clearly articulate the semantic perspective that they target. We offer the RPB framework as a tool for making such learning objectives more explicit, precise, communicable, and heedful of what is known about student learning.

4.2 Conceptual Change and Misconceptions

Research on *conceptual change* seeks to understand changes in how people conceptualize phenomena. Many theoretical models of conceptual change have been proposed; a recent review identified 86 from just five major journals [109]. Some models describe beginners' knowledge as relatively coherent "naïve theories" that are supplanted by better ones as one learns [95, 98, 146]; others characterize knowledge as initially disorganized context-dependent fragments that mature into integrated concepts [33, 98, 146]. A commonality between conceptual-change models is that they emphasize content-specific aspects of learning, the influence of everyday experience on intuitive knowledge, and the difficulty of changing some conceptions through instruction. They aim to help students reach normative models of specific phenomena. RPBs assist in defining that normative model for a particular programming-education context; the RPB framework defers

to the unyielding reality of the computer system (cf. Reference [11]) yet permits a pluralism of semantics that reflect that reality.

Some forms of conceptual learning are relatively easy: adding new knowledge, filling in gaps in existing concepts, and amendment of simple facts. Replacing or restructuring existing knowledge and beliefs is cognitively more demanding [95, 106, 146]. The latter form of learning can be difficult either because the learners are committed to an incorrect but coherent "theory" or because the new information melds into a fluid mishmash of knowledge where normative understandings exist in parallel with non-normative ones [146]. With an RPB progression, a teacher can plan out how new information is gradually added and integrated while minimizing interference and "unlearning."

Students' *misconceptions* about specific programming-language constructs have been documented in dozens of studies across several decades. (For recent reviews, see References [114, 133].) Some misconceptions are syntactic, others semantic; many semantic misconceptions involve the "invisible" aspects of program behavior at runtime, such as the contents of memory, control flow, parameter passing, return values, references, objects, and so on. Learners infer their own "rules" about programming constructs [29, 54], but without sufficient guidance, these rules are often under- or over-generalized. When reasoning about a program, learners may call on a variety of ostensibly contradictory "rules" or notions that they associate with different programs or other contextual cues [20, 27, 85, 94, 133]. Even a so-called misconception is generally not "dead wrong" but a mismatch between knowledge that is productive for some purpose other than the context at hand. Evidence shows [58, 62, 110, 121, 139, 148] that the conceptual difficulties of introductory programming do not vanish even if the syntax barrier is lowered using blocks-based environments.

What counts as a misconception depends on the language and the model of program behavior. For example, the notion that the computer continuously checks the conditions on `ifs` and `whiles` is generally identified as a misconception [36, 102] but it is not universally incorrect. Indeed, Touretzky et al. [142] report the reverse problem: In the Kodu language, the computer *does* check conditions every few milliseconds and trigger behaviors accordingly, so that the sequential thinking employed by some learners is fallacious. (For other examples, see Reference [83].) That being said, many misconceptions have been observed across several similar languages.

We envisage a two-way relationship between RPBs and (mis)conceptions. RPBs set an explicit model against which students' conceptions may be examined. And, on the other hand RPB design can be informed by research on misconceptions and conceptual change.

4.3 Causal Systems and Mental Models

Nelson et al. [97] discuss program comprehension in terms of *causal inferences* [61], highlighting types of knowledge that help people learn efficiently from examples of causal relationships. According to this theory (which we paraphrase heavily from Reference [97]), students should know (1) what *entities* there are in the causal system; (2) how the various entities *depend* on each other; and (3) what *constraints* make behaviors plausible or likely, e.g., computers do not reason as humans do.

Other researchers (e.g., References [133, 145]) have been influenced by theories of *mental models* of causal systems. One such theory is that of de Kleer and Brown [30, 31], who argue that expertise on a causal system is characterized by a *robust* mental model. A robust model consists of submodels of system components; those submodels are modular in that their internal behavior is governed by rules that are independent of other components' internals. Moreover, the rules that specify a component's behavior are independent of what the overall system accomplishes: The *purpose* (or "function") of the whole does not impinge on one's understanding of the system's

structure (cf. also References [76, 124]). Although a non-robust model may work to an extent, a robust model is beneficial for troubleshooting and transfer of learning. Even a single line of code such as `my_number = my_number + 1` may be considered as a causal system with several components. A person with a robust mental model of the system can reason about it independently of the statement's purpose, in terms of the rules that govern its components (i.e., expressions, variables, and assignment). Someone with a non-robust model might have simply memorized this pattern as an atomic "variable-incrementing command."

RPBs can be used to express an expert's robust mental model of program behavior. RPBs may identify system entities and their dependencies; they can also provide constraints on what is plausible within the system. They can characterize program execution in terms of components that each follow certain rules independently of the overall purpose of the program, subprogram, or line of code. We believe, therefore, that RPBs can be useful for specifying which principles learners need to capture in mental models of their own so that they can trace and debug reliably, envision changes to their code, and transfer their knowledge to new programs.

4.4 Notional Machines

In an influential commentary, du Boulay [36] pointed out that one of the major challenges for beginner programmers is that they need to learn a *notional machine*, to wit, "the general properties of the machine that one is learning to control." Those properties are "implied by the constructs in the programming language employed" [37] but may be taught explicitly.

The concept of a notional machine has gained traction in CER since du Boulay's article and perhaps especially in the past decade. It has motivated pedagogies, visualization tools, and research studies. It has been used to interpret empirical findings and cohere separate threads of research [132]. It continues to inspire exciting research questions and influence how the CER field views programming languages [83] and other computer-science topics such as algorithmic complexity [81, 83]. In 2019, a Dagstuhl Seminar was organized under the title *Notional Machines and Programming Language Semantics in Education* [64]. A recent international working group has collected and documented notional machines that are used by teachers [47].

4.4.1 A Term with Many Meanings. The fecundity of the notional-machine concept, probably assisted by its loose definition, has resulted in a proliferation of interpretations. Some authors call educational visualizations notional machines; others distinguish between notional machines and student-facing materials. Some state that only something explicitly written down counts as a notional machine; others use a broader definition where a notional machine may be either explicit or implicitly embodied in a pedagogy or tool. Some assume a single notional machine per programming language; others stress that a single language may be explained in terms of different notional machines. Some have referred to notional machines as mental representations; others have distinguished the concept from them, and so on. It is our impression that all the entities represented by dark blue rectangles in Figure 4—and other things besides—have been called notional machines. In practice, we have often found it difficult to guess at what colleagues and authors precisely mean when they say or write "notional machine."

4.4.2 Notional Machines and RPBs. Our work in this article is intimately linked to the literature on notional machines. We might have adopted one of the notional-machine definitions and built the RPB concept on it. We felt, however, that the existing term was too indistinct and unfixed for present purposes and it would have been counterproductive for us to rely on it. In this article, we discuss the links between RPBs and several related concepts—all of which have been sometimes

called "notional machine"—and we found ourselves unable to do so with clarity except (we hope) by introducing new terminology.[1]

As for whether an RPB ruleset constitutes a notional machine, that depends on which meaning is given to the latter term. For instance, RPBs are not student-facing materials or mental models. RPBs are also never implicit, and so differ from some notional-machine definitions (such as Reference [132]). We do not intend to use this article to debate what should or should not be called a notional machine.

Several recent outgrowths of the notional-machine literature warrant a separate mention here.

First, as part of a far-reaching review, Krishnamurthi and Fisler [83] comment on the similarity of the concept of semantics as discussed by the programming-languages community and the concept of notional machine as discussed by the CER community. If "notional machine" is taken to mean an operational semantics written for educational purposes, then RPBs are a closely related concept, with the proviso that RPBs can be informal and imprecise.[2]

Second, Pollock et al. [107], building on earlier work by Berry [13], conciliate the CER concept of notional machines with *abstract machines* from programming-languages research. Pollock et al. cite the benefits of formal, fine-grained, complete, machine-readable rulesets as a basis for creating versatile educational visualizations and comparing languages in detail. Their work is similar to ours in that it seeks to identify semantic rules on which to build pedagogy and separates those rules from teaching methods. We see formal abstract-machine rules as one style of writing RPBs that has benefits for some purposes, but our framework is broadly inclusive of other styles as well.

Third, Nelson et al. [97] explored how to teach "not the abstract formal semantics for a language, but the semantics as actually implemented in a language's interpreter, mapped to a notional machine to facilitate comprehension" (p. 3). Nelson et al. emphasize, as we do, the need to define and teach the rules of program execution. The thrust of their work was to produce a particular set of rules and a corresponding pedagogy that exemplify comprehension-first programming education, whereas our goal is to sketch out a broader framework for discussing designs such as theirs.

Fourth, Touretzky et al. [141, 142] formulated and studied "laws" that describe what children should learn about the behavior of programs created in an event-based programming environment for beginners. The laws, which can be taught explicitly or via guided discovery, help learners reason about code, predict behavior, and generally appreciate the "lawfulness" [141] of computer programs. The authors have linked their work to research on tracing and notional machines [142]; we moreover interpret it as an instance of the rule-based approach that we are proposing here.

Fifth, Dickson et al. [32] seek to make the notional-machine concept more accessible to teaching practitioners. The authors delineate between notional machines, mental models, and visualizations, arguing that if teachers distinguish between these concepts, then they are better positioned to make use of each. Dickson et al. moreover demonstrate how a single notional machine may be visualized in different ways and, conversely, how the same visuals can be applied to multiple notional machines. We advance a similar agenda and expand on it: Our RPB framework highlights the conceptual distinctions that Dickson et al. mention and is intended to help teachers and researchers use these concepts and communicate about them.

Sixth, Fincher et al. [47] define notional machines as student-facing pedagogic devices, each with a particular representation such as a metaphor or a visualization. Moreover, they interviewed teachers and studied educational materials to catalogue such notional machines. In RPB terms, Fincher and colleagues' pedagogic devices are teaching methods that target various models of

[1]We decided not to use either "notional" or "machine" in the name of our framework to avoid any suggestion that RPBs are mental notions or that they are (primarily) about hardware.

[2]We chose "rules of program behavior" over, say, "education-oriented operational semantics" to distinguish this educational framework from research in programming languages and, perhaps, to make it easier to communicate to a wider audience.

program behavior (or fragments thereof). Our focus in this article is on describing such models as RPBs, not on the student-facing materials.

Seventh, and finally, Greg Wilson's blog post *Is This a Notional Machine for Python?* [150] informally describes an execution model for Python programs as 14 bullet points. It inspired discussions at the 2019 Dagstuhl seminar on notional machines [64]; the present work is an offshoot of those discussions. We thus consider Wilson's list the prototypical RPB ruleset *avant la lettre*.

In summary, our work echoes the advice from the notional-machine literature to acknowledge semantical models as key learning objectives (cf. e.g., References [32, 36, 83, 118, 132]). We believe that RPBs complement the extant literature as a tool that (1) helps treat models of program behavior—and progressions of such models—as explicit learning objectives; (2) provides a framework for discussing, documenting, and evaluating designs; (3) brings focus to meaningful research questions; and (4) might assist in distinguishing between the meanings attached to "notional machine."

4.5 Sublanguages and Language Progressions

Programming teachers and textbooks introduce language features in progressions. The appropriate ordering has been debated for decades. Whichever order is chosen, the common approach is to gradually expand coverage of a language, introducing new syntax and the corresponding behaviors together. Some new syntactic features can be explained in terms of earlier knowledge of program behavior, while other additions demand a more complex model.

Many experienced teachers consider such matters when designing their courses, but documenting a progression of semantic rulesets is not a common practice. The most prominent exception to the rule is Racket, a language that "is about creating new programming languages" [45]. Racket is supported by the DrRacket environment [49, 50] and often used in conjunction with the *How to Design Programs* curriculum [43, 44]. In this curriculum, Racket is introduced as a sequence of increasingly sophisticated *sublanguages* that the environment is sensitive to. A sublanguage is a subset of a programming language that is implemented as a language in its own right. Working within the confines of a sublanguage brings a variety of benefits, such as better error messages, elimination of confusing encounters with unfamiliar language features, reduced risk of student programs that "work" by accident, learner-friendlier documentation, and so on [50, 83]. An especially germane feature of sublanguage progressions is that sublanguages can differ from the full language (and each other) not only syntactically but also semantically: "One particularly useful design criterion [for sublanguages] is to layer the complexity of the notional machine" [83, p. 398].

Although the sublanguages approach is not the mainstream, the idea has existed since the 1970s [73] and been applied to several languages [6, 9, 16, 19, 59, 140]. A recent entry in this space is Hedy [70], a language designed for incremental introduction, primarily to ease beginners' difficulties with syntax. Outside of introductory programming and formal education, sublanguages have been suggested as a way to ease industry professionals' migration between languages [123].

The RPB framework is compatible with the sublanguages approach but distinct from it.

The work on sublanguages pursues the ideal that, at each stage of a curriculum, learners know a semantics that is viable for any program in the (sub)language that they currently use. This work points at the need to create better teaching languages, compilers, and programming environments. Using our terminology, sublanguages are a way to implement an RPB progression.

In contrast to a typical sublanguage semantics, an RPB ruleset will often cover a very limited part of a language. There is no implication that each ruleset must be matched by a restricted language or bespoke tooling—although it *may* be. Despite the unique benefits of sublanguages, we want our framework to cover the manifold contexts that depend on a programming language without sublanguages. (This is to help more people like Amal, Bao, Camille, and Dana from the Prelude.)

4.6 Teaching Methods

The RPB framework does not prescribe teaching methods. On the contrary, it highlights the distinction between educational goals (i.e., RPBs) and the methods for achieving those goals.

As we have already mentioned, it is uncommon for introductory programming curricula to articulate a model of program behavior explicitly. However, many teaching methods and tools implicitly target some such model, whose definition may be vague or nonexistent.

The sheer number of methods makes it difficult for teachers to choose between them. The enormous variety between the methods makes it hard for teachers and researchers to compare them. The vagueness of the targeted models of program behavior makes evaluations and comparisons harder still. We suggest that when comparing teaching methods, specific learning objectives regarding program behavior should be part of the discussion.

A full review of teaching methods is beyond the range of this article. However, this article's online supplement in the ACM Digital Library [40] lists teaching methods that have direct connections to the RPB framework. These include the following: tracing and debugging activities; program visualizations; microworlds; tangible computing; serious games, analogies, and metaphors; and strategies for topic sequencing. Either explicitly or implicitly, all these methods target improvements to how learners perceive program execution. However, they target many different models of program behavior; if we had RPB rulesets to match each method, then it would be a varied assortment.

Whether any given method "works" depends on specific objectives. We argue that RPBs can help instructional designers and researchers analyze programming pedagogies by differentiating between learning objectives and teaching methods. And even when a method or tool has already been deemed useful, RPBs can help communicate it and analyze its relationship to other methods, such as when designing for transfer from a programming language to another or from a microworld to a more generic programming environment. RPBs may also be used as a foundation for the design of new visualizations, games, and other learning activities.

We will return to these potential benefits of RPBs under Discussion (Section 7) after presenting our case-study RPBs for Python programming and considering RPB design.

5 EXAMPLE: AN RPB PROGRESSION FOR PYTHON BASICS

In this section, we present a local progression of RPB rulesets, each accompanied by code that exemplifies programs that the ruleset is designed for. The first ruleset appeared above in Figure 3; the rest are in Figures 5–8. Our five rulesets are designed for the early stages of an introductory university course that uses the Python language. The target course does not in fact exist exactly as described: What we present here is a close adaptation from the pedagogy of an actual course designed and taught by the second author, which uses a different, statically typed language.

Our RPB progression is meant for programs that feature imperative commands, mutable state, and aliasing. It does not feature objects in the OOP sense but is designed to support the introduction of object-oriented concepts immediately after these five rulesets have been introduced and sufficiently practiced on. Selection and repetition also come up later and are beyond the scope of these rulesets.

The course attempts to teach a viable model of program behavior so that students will write code that they can themselves understand, as opposed to twiddling with example code and perhaps getting a program to work by accident. There is an article [137] that describes some of the principles that have guided the course's instructional design, and the electronic textbook that shapes the course can be viewed online [135].

§9 A *variable* is a location in computer memory that has a *name*. You can create and name variables in code.

§10 You can *assign* to a variable to store a value in memory.

§11 You can use a variable's name as an expression. The computer evaluates the name by retrieving from memory the value currently stored in the variable.

- §11.1 Since a variable name is an expression, you can use it as part of a composite expression.
- §11.2 *The name's natural-language meaning is irrelevant to the computer.

§12 A variable must be assigned a value upon creation.

- §12.1 A variable never remains empty.

§13 The computer deals with assignment by taking an expression ('on the right') and storing its value in the target variable.

- §13.1 Once the computer has the expression's value in the evaluation area, it copies the value into the variable.
- §13.2 *It's the value that's stored, not the expression.
- §13.3 You can use any expression 'on the right': arithmetic, literal, variable name, etc. Assignment works the same anyway: first evaluate, then store the result.
- §13.4 *An assignment command is not symmetric. Even `a = b` does not have a variable on the left and a variable on the right in the same sense: `b` is an expression and `a =` indicates the target location.

§14 A variable stores a single value at a time.

- §14.1 Whatever was in the variable upon assignment is discarded.
- §14.2 A variable may or may not keep storing the same value, depending on whether the code reassigns to it.
- §14.3 *A variable does not store multiple values, or a history of its values, or a sum of the assigned values.
- §14.4 *Assignment is not an equation. It does not 'forever equate the symbols.' `a = b` or `a = b + 1` does not persistently link variables together.

§15 The computer runs commands in sequence. Previously executed commands can affect what later ones do.

- §15.1 *The computer does not look at the lines 'as a whole' or continuously keep checking each command.

§16 Where a variable name appears as an expression, it evaluates to what is in the variable *at the moment of evaluation.*

- §16.1 *A sequence of assignment commands does not get 'solved' like math equations.
- §16.2 *Code like `a = a + x` is not special. It works like any other assignment: evaluate, then store.
- §16.3 *Assigning to a variable V does not instantly impact on the value of any other variable, not even if V has previously appeared on the right-hand side of an assignment statement.
- §16.4 You cannot access a variable that has not been created.

Example code:

```
>>> factorial = 1 * 2 * 3 * 4 * 5 * 6
[evaluation and storage not shown by REPL]
>>> factorial * factorial * factorial
373248000
```

```
>>> my_test = 100
>>> another = 1 + my_test
>>> third = another
[effects not shown by Python REPL]
>>> 1 + my_test
101
>>> my_test - third
-1
```

```
>>> number = 10
>>> twice_that = number * 2
[effects not shown by Python REPL]
>>> twice_that
20
>>> number = 3
[effect not shown by Python REPL]
>>> twice_that
20
>>> twice_that = number * 2
[effect not shown by Python REPL]
>>> twice_that
6
```

```
>>> coordinate = 3
>>> velocity = 5
>>> coordinate = coordinate + 1
>>> coordinate = coordinate + velocity
[effects not shown by Python REPL]
>>> coordinate
9
```

```
>>> temp = first
>>> first = second
>>> second = temp
[effects not shown by Python REPL]
```

```
>>> asdf
NameError: name 'asdf' is not defined
```

Fig. 5. Example ruleset #2: Variables, mutable memory, and sequencing. The "negative" items marked with an asterisk directly contradict documented student misconceptions.

We are not constructing an argument here for the superiority of this progression over any other. We mean to give an example of what an RPB progression may look like and to provide a point of reference for the RPB design criteria in Section 6 below.

To keep this section shorter and simpler, we have omitted some concepts and constructs. The greatest omission concerns data types: All our examples here deal with integers and lists of integers, even though the actual course uses a combination of integers, floats, strings, and immutable `Pics` and `Colors` from a library right from the start. Another is that we have left out the `print` command and library functions for playing sounds and displaying pictures (all of which the course introduces as special cases before other functions). Our example code consists only of decontextualized fragments that suffice for present purposes, which belies how the course uses both decontextualized code and considerably more interesting programs that employ the same constructs.

§17 A *list* is a collection of *elements* in computer memory.
- §17.1 A list may contain one or more elements or be empty.
- §17.2 A list's contents may change. A list's size may change.
- §17.3 A list's elements are numbered with *indices*.
- §17.4 Lists exist in memory outside of any variable or evaluation area.

§18 There is an expression that you can write to create a list. Such an expression evaluates to a new *reference*.

§19 A reference is a value that indicates the location of certain data in memory, such as a list.
- §19.1 A reference is essentially a number that identifies a memory location. The computer uses that number behind the scenes. The programmer doesn't see it in code and doesn't have to worry about the exact number.
- §19.2 A reference, being a value, can be stored in a variable.
- §19.3 *The name of a variable is still just that. References or lists do not have names.

§20 You can access a specific list element by writing an appropriate expression. The expression must specify the reference to the list and the index.
- §20.1 The reference can come from any (sub)expression that evaluates to a list reference (e.g., a variable name); the index from any `int`-valued expression (not only name or literal).

§21 You similarly use a reference and an index when you modify an element, or add or remove elements.

§22 If you use too high an index, you get a runtime error.

§23 You can use a reference-valued expression 'on the right,' as in `my_ref = your_ref`.
- §23.1 The source expression evaluates to a reference. That reference is copied into the target variable.
- §23.2 *Such a command does not mean copying the contents of the referenced object (list). It does not 'rename' the object.

§24 If you reassign a variable to store a different (list) reference, a different (list) object becomes accessible via that variable.
- §24.1 Mutating a list's contents is different from mutating which variable(s) refer to the list.

§25 Variables can store identical references. For example, multiple variables can point to the same list, making it accessible via different variable names.
- §25.1 If you mutate a list through one such variable, that change will be observable via other variables as well.

Example code:

```
>>> numbers = [12, 2, 4, 4, 7, 4, 10, 3]
[effect not shown by Python REPL]
>>> numbers[0]
12
>>> fifth_element = numbers[4]
>>> numbers[4] = 100
[effects not shown by Python REPL]
>>> numbers
[12, 2, 4, 4, 100, 4, 10, 3]
>>> fifth_element
7
>>> 1000 + fifth_element + numbers[0]
1019
>>> numbers[10]
IndexError: list index out of range
```

```
>>> my_list = [4, 10, 3, 10, 15, -2]
>>> some_index = 2
[effects not shown by Python REPL]
>>> my_list[some_index]
3
>>> my_list[some_index + 1]
10
>>> my_list[some_index] + 1
4
```

```
>>> my_list = [1, 2, 3, 4]
>>> my_list[2] = 999
>>> my_list = [11, 22, 33, 44]
>>> my_list[0] = -1
[effects not shown by Python REPL]
>>> my_list
[-1, 22, 33, 44]
```

```
>>> first = [1000, 100, 10, 1]
>>> second = first
>>> first[2] = 999
>>> second[0] = -1
[effects not shown by Python REPL]
>>> first
[-1, 100, 999, 1]
>>> second
[-1, 100, 999, 1]
```

Fig. 6. Example ruleset #3: Lists, references, and aliasing.

Ruleset #1 in Figure 3 is focused on the concepts of expression and evaluation. These concepts are introduced not because they are strictly necessary for explaining the programs that students read and write at this stage but because they pave the way for the rulesets and programs that follow. The second ruleset in Figure 5 extends the first by adding the concept of variable. (All our example rulesets subsume the preceding rules without modification.) The third ruleset in Figure 6 further extends the first two by introducing mutable collections that are accessible via references. These concepts will assist learners as they encounter stateful functions in Rulesets #4 and #5 and object-oriented programs with mutable state soon afterwards. Those last two rulesets (Figures 7 and 8) deal with calling and implementing functions, respectively.

During the early stages that introduce these rulesets, the REPL is the students' primary programming environment. Students explore expressions, variables, function calls, and so on, in the REPL. Once function implementations (Ruleset #5) come into play, students write their own functions in files and call the functions via the REPL. Application entry ("main") is introduced later in the course and not covered here.

§26 A *function* is a program component that takes care of a particular task.
§27 You don't need to know a function's implementation in order to make it run.
  §27.1 You can run functions from *libraries*, which may be part of the system or provided by a teacher or others.
§28 To run a function, you write an expression that *calls* it.
  §28.1 The computer evaluates the function-calling expression by executing the code that implements the function.
§29 A function-calling expression specifies the function to call and some number of *arguments*.
  §29.1 You write argument *expressions*, which the computer evaluates as per usual before running the function.
  §29.2 The resulting argument *values* are passed to the function. The function's implementation may use those values.
  §29.3 *An argument expression does not have to be a literal or a variable name.
§30 A function-calling expression evaluates to the function's *return value*.
  §30.1 Functions always return a value.
  §30.2 However, a function may return the special value `None`, which means there is no return value of interest.
  §30.3 *You don't have to assign the return value to a variable. You can do anything with it: assign it, do arithmetic on it, etc. Or leave it for the REPL to print out.
  §30.4 *Returning a value is different from printing it out.
§31 A function may or may not have observable (side)*effects* on output or stored state (e.g., a list's elements).
  §31.1 Some functions never do and are similar to math functions: arguments in, return value out.
  §31.2 *But in general functions are *not* like (the usual) math functions: they specify computations by the machine, sometimes with observable effects.
§32 Function-calling expressions are expressions too.
  §32.1 You can combine them in arithmetic expressions and other function-calling expressions.
  §32.2 A function call written into another call's argument expression will run before the surrounding call.

Example code:

```
>>> data = [-2, 0, 10, -100, 50, 100, 5, -5, 2]
[effect not shown by Python REPL]
>>> remove_negatives(data)     # teacher-provided lib
[effect and return value of None not shown]
>>> data
[0, 10, 50, 100, 5, 2]
```

```
>>> average(5, 11)             # teacher-provided lib
8
>>> average(13, 7) + 100
110
>>> first = 10 + average(1, 11)
[effect not shown by Python REPL]
>>> 2 + average(first + 3, 1)
12
>>> average(24, first) + average(0, -10) - 1
14
```

```
>>> min(10, abs(-15))                # standard lib
10
```

```
>>> do_sthg_fancy_that_i_cannot_implement_yet(args)
```

Fig. 7. Example ruleset #4: Calling functions.

Together, the rulesets show that understanding the behavior of ostensibly simple code requires learning a number of concepts, making various fine distinctions, and avoiding a number of pitfalls. The RPBs detail dozens of micro-level learning goals that the course's teachers have identified as relevant and hope to attend to.

These RPBs, like others, might be taught (or implicitly acquired by students) in any number of ways. In the actual course, we use a combination of activities, code examples, textual descriptions, program visualizations, and concept maps [135, 137].

6 DESIGNING RPB RULESETS AND PROGRESSIONS

In this section, we first identify factors that feed into the design of models of program behavior and the RPBs that describe those models (Section 6.1). The bulk of this section is devoted to criteria for evaluating and comparing RPB designs (Section 6.2).

6.1 Factors Contributing to RPB Design

The main requirement for writing an RPB ruleset is to identify the model of program behavior that the RPBs express. The diagram in Figure 9 illustrates the knowledge that informs this design. Below, we briefly discuss each of the four inputs at the top of the diagram: knowledge about the computing *system*, the *educational context*, the *learners*, and content-specific *pedagogy*.

6.1.1 Knowledge about the System. RPB design is grounded in the system whose behavior is being taught. That system comprises the programming language, runtime environment, and other tooling such as compilers. Formal semantics may inform RPB design (cf. Reference [107]), as may the implementations of programming-language interpreters (cf. Reference [97]). RPBs may

§33 You can write a function definition in the REPL or save it in a file.
- §33.2 *When you define a new function, its code does not run right away. It runs whenever you call it.

§34 A function's code dictates the computer's behavior when it activates the function, but a function activation is distinct from the code.

§35 The computer keeps track of a function activation in a *frame* in memory.
- §35.1 Each frame has a separate evaluation area for evaluating expressions while running the function body.
- §35.2 The frame also holds any *local variables* that are defined within the function body and exist temporarily during the function activation.
- §35.3 The initial calling context is also a frame: it has its own evaluation area and local variables.

§36 The frame exists while the function is active.
- §36.1 When an activation starts, the computer creates a new frame for it 'on top of' the calling frame. When the activation terminates, the computer removes the frame.
- §36.2 In the meantime, the calling frame remains suspended.

§37 Within a function body, you may write expressions that call functions.
- §37.1 Evaluating those calls adds further frames 'on top of' the *call stack*.

§38 *Parameter variables* (formal parameters) are a special kind of local variable.
- §38.1 You don't assign to them explicitly. They receive the values of the evaluated arguments as soon as the function activates. The computer copies argument values from the calling frame's evaluation area into the new frame.
- §38.2 *Even if the argument expression is a variable name, the computer just copies its value into the other frame. It does not link the variables in any way.

§39 Once the parameter variables have received their values, the computer begins running the called function's body. It executes the commands in the body in order, until a value is returned.
- §39.1 One way to return a value is a direct command to evaluate an expression and return its value. This terminates the function activation.
- §39.2 *The return value does not need to come from a variable; any expression will do.
- §39.3 Another way to return a value is to reach the end of the function body without encountering a return command. In this case, the computer defaults to returning `None`.
- §39.4 *The computer does not print out the value of each consecutive expression within the function body (cf. the REPL).

§40 When a function activation terminates, the return value becomes the value of the calling expression.
- §40.1 The computer copies the return value from the expiring frame's evaluation area into the evaluation area of the frame below.

§41 Execution then resumes in the calling context. What happens next depends on what command the function-calling expression is part of.

§42 In addition to parameter variables, a function body may create additional local variables.
- §42.1 They, too, exist only temporarily within the frame.
- §42.2 Local variables can only be used within the function body that they are local to.

§43 Local variables are distinct from variables in other frames.
- §43.1 *Even having the same name as a variable in another frame does not connect the variables or mean they are the same.
- §43.2 However, variables in different frames may store references to the same (list) object. Mutations via one such variable are observable via others.

Example code:

```
def average(first, second):
    return (first + second) / 2
```

```
def print_twice(message):
    print(message)
    print(message)
```

```
def set_first_to_zero(list):
    list[0] = 0
```

```
def tax(income, threshold, base_rate, extra_rate):
    base = min(threshold, income)
    extra = max(income - threshold, 0)
    return base * base_rate + extra * extra_rate
```

```
def distance(x1, y1, x2, y2):
    return hypot(x2 - x1, y2 - y1)

def greatest_distance(x1, y1, x2, y2, x3, y3):
    first  = distance(x1, y1, x2, y2)
    second = distance(x1, y1, x3, y3)
    third  = distance(x2, y2, x3, y3)
    return max(first, second, third)
```

```
def swap_contents(a, b):
    temporary_store = a
    a = b
    b = temporary_store

swap_contents(my_var1, my_var2) # No effect.
swap_contents(a, b)             # Even this fails.
```

Fig. 8. Example ruleset #5: Function implementations, parameters, and the call stack.

highlight the interplay between program runs and the environment that learners use for writing programs, such as an IDE.

Software libraries can greatly impact RPB design, as they affect which concepts the learners have to understand and which ones can be abstracted away for the time being. Programming languages' standard libraries are relevant, as are other libraries, such as microworld interfaces and other scaffolds for learning. As an example, the repetition of operations on collection elements might be explained in terms of very different RPBs depending on whether the learners use state-based loops (such as `while`) or library functions on collections (such as `map` or `filter`).

6.1.2 Knowledge about the Educational Context. The quality of RPBs depends on the goals of the course or module they are intended for. If that course is part of a broader curriculum such as

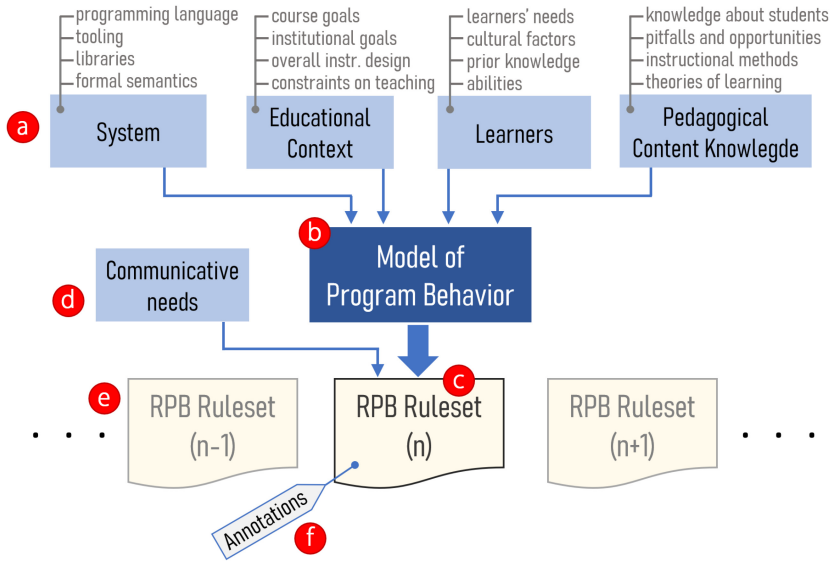

Fig. 9. Several types of knowledge *(a)* feed into the design of a model of program behavior *(b)*. That model can be expressed as RPBs *(c)* to meet some communicative needs *(d)*. The resulting RPB ruleset may be part of a progression *(e)* and may be accompanied by information about the design choices that produced it *(f)*.

a degree programme, then those curricular goals also need to be considered. For example, a short course that merely gives a taste of programming and sparks interest may rely on a simpler model of program behavior than a foundational course for software-engineering majors. If students learn programming conjointly with another subject (e.g., math, crafts, data science), then some RPBs may synergize better with that subject than others [83]. In some courses, it can be appropriate to gloss over performance, while others require a model that enables discussions of efficiency. A course that emphasizes security in C programming will need a different model than one that does not.

Moreover, an RPB ruleset may be intended for a particular stage of a course so that its design is influenced by where it fits in the course's overall instructional design.

RPB design also needs to be mindful of restrictions imposed by the educational context, such as the need to meet specific standards, the lack of available resources (e.g., time, teachers, computers, network connections), or other constraints on what educational goals are realistic.

6.1.3 Knowledge about Learners. The design of RPBs, like any learning objectives, is crucially influenced by the intended audience. To begin with, the generic things matter: What is the learners' age and stage of cognitive development? Are their study skills or metacognitive skills known? Are they literate and numerate? Do they have disabilities?

Culture matters. Are there tensions between the learners' cultures and the type of programming they are learning? Are the RPBs intended to align with that culture or to disrupt it?

Language matters. Are there natural-language challenges [10, 63, 115] that limit the effectiveness of teaching to the extent that goals (RPBs) need adjustment?

Prior knowledge in computing matters crucially. What a learner already knows affects which parts of an RPB ruleset are new to them, how hard those rules are for them to learn and recall, and what they can reach in a given time frame. Prior knowledge shapes how learners interpret new information, which influences not only which teaching methods are appropriate but also the specifics of what to teach. For example, students with prior knowledge of assembly programming

Cultural fit	Consistency	Notational fit	Wieldiness
Accuracy	Granularity	Generality	Assessability
Coverage	Abstraction	Transferability	Implementability
Simplicity	Expressibility	Sensitivity to conceptions	Clarity of writing

Fig. 10. Criteria for evaluating tradeoffs in models of program behavior.

will perceive any new model differently than other students who have only programmed in Scratch or have no prior programming experience. RPBs may be designed to highlight the differences to previously learned models, as those differences may challenge learners; on the flip side, RPBs may explicitly link to the knowledge that learners are expected to have, and leverage it.

6.1.4 Pedagogical Content Knowledge. As mentioned in Section 3, RPBs incorporate the designer's pedagogical expertise. The effort of RPB design is influenced by the designer's awareness of common pitfalls, methods for teaching particular content, and other content-specific pedagogical knowledge.

Knowledge of learners' likely (mis)conceptions is a significant component of pedagogical content knowledge [113, 119, 151]. An especially direct way to apply pedagogical content knowledge in RPBs is to include items such as the asterisked ones in our examples, which directly contradict potential misconceptions (see, e.g., Figure 5). That is, if a non-normative conception is prominent enough, then avoiding or improving it can be worth noting as a learning objective.

Phrasing and annotating RPBs. RPBs' wording depends on who they are written for—only their author, colleagues in the same context, or other teachers and researchers?

In addition to writing down the rules of a ruleset, the author may wish to annotate them with comments that document the intended context, the design decisions that produced the RPBs, or other relevant information that is not part of the RPBs *per se*.

6.2 Evaluation Criteria

RPBs express detailed learning objectives that are subservient to broader objectives. The overarching criterion for judging any RPBs is how well they serve that purpose, which is a multi-faceted question. Figure 10 lists 16 criteria for evaluating models of program behavior and the RPBs that describe such models; they are interdependent in a complex way. Our list is non-exhaustive and ordered for ease of exposition, not by importance. It is a synthesis of our own thinking with suggestions from the literature on notional machines and similar concepts; in broad terms, the discussion is influenced especially by cognitive theories of learning. We intend it as a tool for discussing tradeoffs in RPB design, perhaps similarly to how Green and Petre's *cognitive dimensions* [60] can be used to debate the relative merits of notations.

6.2.1 Cultural Fit. As discussed in Section 6.1 above, RPB design is influenced by cultural factors from the educational context and the learners' backgrounds and goals. For example, there may be synergies and tensions between computing and other school subjects. The criterion of cultural fit refers to how well RPBs attend to cultural and cross-disciplinary considerations.

6.2.2 Accuracy. This is the RPB cousin of the formal-semantics concept of soundness. An RPB ruleset is accurate if what the rules say about the system effectively happens and the rules thus lead to correct predictions of the system's behavior. That ground truth is established by the programming language and the system's implementation.

Accuracy is obviously desirable. However, some inaccuracy may be welcome for a variety of reasons, such as simplicity or expressibility. Teachers are familiar with this compromise: "Well, that's not *exactly* how it (always) works, but you can think about it this way for now." An RPB might describe floating-point numbers as behaving like decimal numbers, for example, or Java variables as containing the characters of a `String`. Even the idea of imperative statements being executed strictly one after the other is inaccurate in modern computers, but the illusion of pure sequentiality is good enough for many purposes.

Like the other criteria, the need for accuracy depends on context. RPBs may be designed for short-term needs: "If the goal is to . . . allow them to solve simple, immediate problems, then . . . a deep understanding of the semantics of the language is less important" [77]. However, if the goal is deep learning in the long term, then it is a problem if "students develop only a vague and inaccurate understanding of basic language constructs and fundamental programming techniques" (ibid.).

Phrasing RPBs metaphorically might lower accuracy but be a useful shorthand. Moreover, we distinguish between using metaphor in RPBs (i.e., in the teacher-facing objectives) and using metaphor when teaching the RPBs to students. The *semantic waves* approach to lesson design [26, 147] exploits a temporary dip in accuracy: A brief technical introduction is followed by metaphors and analogies before the lessons learned are deliberately linked back to the technical concepts.

6.2.3 Coverage. An RPB ruleset may be viable for only some of the programs that can be written in the learners' programming language. The more programs it is viable for, the higher its COVERAGE. Coverage is a cousin of completeness in semantics.

Incomplete coverage means that learners may encounter or write programs whose behavior the RPBs do not explain, or explain incorrectly. Error messages are a noteworthy part of behavior; incomplete coverage means, among other things, that error messages may use unfamiliar concepts.

Various authors have highlighted the need for complete coverage (e.g., References [14, 97, 107]). Completeness is particularly prized in sublanguages approaches (Section 4.5): Each sublanguage along the progression has its own complete semantics and tooling, which shield learners from undesirable surprises. Such benefits notwithstanding, many curricula rely on programming languages for which no sublanguages are available. Where the programming language is too complex for the learners to tackle in its entirety, teachers accept compromises to coverage, increasing it gradually as the learners progress and attempting to mitigate the consequent issues.

6.2.4 Simplicity. An RPB ruleset is simpler if it has fewer concepts and—especially—fewer complicated relationships between concepts. Beginners' need for a simple model of program behavior has been noted since the early days of CER: "the notional machine should be simple. That is, it should consist of a small number of parts that interact in ways that can be easily understood" [37, p. 265]. As others before [37, 77, 83, 120], we distinguish between conceptual SIMPLICITY of program behavior and syntactic simplicity; we focus on the former.

When designing progressions, simplicity can be considered relative to earlier RPB rulesets and prior knowledge: What rules, concepts, and relationships are new here? Which rules are the learners already fluent with and can "chunk away" or even reason about instinctively? As an example, students might practice on Boolean expressions before selection is introduced. Our rulesets on functions (Figures 7 and 8) are another example: They follow the *consume-before-produce* principle [18, 101] of students using an abstraction before they learn to implement it.

Various arguments have been made in the literature concerning simplicity of different languages and paradigms. Sajaniemi and Kuittinen [120] advocate procedural over object-oriented programming: "In contrast to the procedural approach, OOP requires a much larger and more complicated notional machine from the very beginning." Felleisen et al. [43] similarly point at added semantic complexity from objects compared to functional programming. Ben-Ari [11], having argued

that a model of a computer must be explicitly taught and that the chosen programming language must not spoil the model's simplicity, recommends that "introductory CSE should be based on the functional or logic programming paradigm, [primarily] because the underlying models can be explained in relatively high-level, hardware-free terms." Krishnamurthi and Fisler [83] suggest starting beginners with a simple notional machine based on immutable state and introducing concepts such as mutation "only after students have gained familiarity with basic programming techniques." Johnson et al. [77] argue that Python's syntactic simplicity belies a complex notional machine.

Many of the other considerations on our list impact on SIMPLICITY. For example, CONSISTENCY and ABSTRACTION tend to make rulesets simpler, whereas COVERAGE, ACCURACY, and fine GRANULARITY may add complexity. For some additional comments on these tradeoffs, see Dickson et al. [32].

6.2.5 CONSISTENCY. CONSISTENCY means few exceptions, few special cases, few unpleasant surprises. "Special cases increase the amount that has to be learned by the novice by complicating the properties of the notional machine" [37]. For example, a ruleset where "everything is an object" has greater consistency than one that separates objects and primitives. A ruleset that treats all assignment statements the same is more consistent than one that treats numerical assignment statements differently than assignment statements that "name lists" (see also GRANULARITY, below).

Consistency helps RPBs provide conceptual constraints on what is plausible or likely within the system; such constraints support learning from example programs (see Reference [97] and Section 4.3 above). If an RPB ruleset is consistent, then new rules seem like a logical extension of what was known before and may be guessable intuitively.

Providing multiple perspectives to program behavior (e.g., a multi-paradigmatic RPB ruleset or many rulesets for many perspectives) is a powerful notion. A caveat is that multiple perspectives can compromise consistency if learners are unsure which perspective is useful for which purpose. Ko [80] and Hermans [69] describe Franke's comments on how a "teachable" language needs a "consistent narrative [to] help learners build larger models about languages, which help them make predictions about how to use them, help them generalize knowledge, and help them retrieve resources about the language" [80]. Seen from this angle, odd language quirks, multi-paradigm languages, and alternative ways of expressing a solution pose challenges that teachers need to work around, which is why some teachers prefer simple micro-languages [69, 80]. This consideration applies to syntax, idioms, and libraries—and RPBs, too. Teachers need to weigh the benefits of multiple perspectives, authenticity and TRANSFERABILITY against those of consistent narratives.

6.2.6 Granularity. COVERAGE concerns the breadth of RPB rulesets. GRANULARITY is about depth: Is program behavior covered in sufficient detail?

Models of program behavior emphasize mechanistic execution: There are certain rules that are always followed, no matter what the program's or subprogram's or line's intended purpose is. (See Section 4.3 on causal systems above.) To capture those rules, RPBs need to "zoom in" from the teleological level that involves the programmer's goals and plans to a structural level where the rules apply mechanically (cf. References [76, 124]). Fine granularity is especially important when learners trace, debug, or modify code in which constructs appear in unfamiliar or flawed patterns.

Ideally, RPBs are detailed enough to explain many combinations of constructs, not only the commonest or simplest patterns. A granular model can reveal connections that might otherwise be overlooked. As an example, we will discuss an issue that matters to many RPB designs: the concepts of *expression*, *evaluation*, and *value*. How useful is it for students to know these concepts and their relationships to other constructs, such as variables, parameter passing, and return values?

Some of the research literature focuses on statements. For example, the Block Model of program comprehension [76, 124] treats statements (rather than expressions at the sub-statement level)

as the "atoms" of program execution and the lowest level for discussing the purpose of language elements. We also observe (anecdotally) that many programming teachers and introductory textbooks do not emphasize expressions and evaluation, except when it comes to arithmetic and logic; it seems common to reserve the word "expression" for expressions constructed using infix operators. Statements such as `num = 10`, `a = b`, `nums = [3, 1, 3]`, or `print(my_func(100))` are often not taught as involving expressions and evaluation. We have heard teachers argue that it is simpler not to; they might prefer to teach `nums = [3, 1, 3]` as "naming a list," for example. Of course, there are also many teachers who do discuss such code in terms of expressions. Sorva [134] recommends:

> *The lines* `result = 2 * (input + 10)`, `i = i + 1`, *and* `myList = list(5, 1, 2)` *have different purposes . . . Students may think of them variously as a "result-computing assignment statement," a "counter-incrementing command," and a "command for naming lists," respectively. The lines' structural similarity may escape the students' attention: Each evaluates an expression and assigns its value to a variable. . . . These key concepts enable students to reason about code in terms of its structural components while setting aside the purpose of the entire line or block of code . . . Don't neglect them.*

In terms of the present article, we can restate that advice as "Teach RPBs that have sufficient GRANULARITY to explain expression evaluation."[3]

In addition to CONSISTENCY, granularity can synergize well with SENSITIVITY TO CONCEPTIONS and ASSESSABILITY. However, fine granularity can compromise SIMPLICITY and WIELDINESS and may be prevented by a high level of ABSTRACTION.

These tradeoffs in granularity are mirrored in the design choices of **program visualization (PV)** systems that illustrate a model of program behavior to novices. Specifically, whether program behavior should be illustrated at a coarser line level or a finer expression level continues to be debated by PV designers. Line-based PV has been the norm [136], but expression-level PV is also supported by a number of tools and appears to be increasingly common [2, 7, 12, 21, 78, 97, 107, 128, 129]. Some authors of line-based PV tools have sought alternative means to highlight evaluation [82], while others have explored unplugged visualizations at the expression level [35, 144]. There is some empirical evidence that students value the expression-level aspect of PV [1, 78] and at least one experiment whose results support teaching program behavior at the expression level [78]. Overall, we interpret the PV literature as suggesting that granularity is a significant criterion in RPB design, whether the rules are taught using PV or otherwise.

6.2.7 Abstraction. This is the other side of the GRANULARITY coin: Has enough unnecessarily detail been eliminated? ABSTRACTION is closely associated with SIMPLICITY.

Many authors have noted the need to find a suitably high level of abstraction. In early work, du Boulay et al. [37] built on Mayer's concept of "transactions" that describe program behavior:

> *[Transactions characterize] a mechanism that has sufficient structure to explain the sequence of events while [a program] is running, but is simple enough to be grasped by a novice and avoids over-technical descriptions that would be confusing and irrelevant. Mayer's transactions are 'black boxes' whose own internal workings do not need to be explained.*

Leaving out detail from a notional machine (or RPB ruleset) is not even optional:

> *Some teachers [hold] the view that students need to understand what 'really happens' [but even] discussions about assembly language or machine code are almost necessarily abstractions, since hardware optimisations of modern processors are so complex . . . A meaningful discussion about notional machines does not centre around the question whether or not to use one, but around the most useful level of abstraction to aim for* [14].

[3]To be clear, we do not mean that a "whole-line interpretation" (e.g., "a variable-incrementing command") cannot be productive (cf. Reference [27]). We merely stress that sometimes students need to look at the finer structure as well.

The design question becomes the following: What can you leave out from an RPB ruleset?

Microworlds and educational software libraries provide abstract interfaces to behaviors that are not topical for the learners. For example, Touretzky et al. [141] observed that the rules for their Kodu microworld did not require subscripted data structures, as they can be expressed using abstract pattern-matching behaviors.

Many models and visualizations of program behavior make computer memory explicit. However, Krishnamurthi and Fisler [83] point out that memory layout may be abstracted away until mutation and aliasing are covered later. They also identify additional opportunities for abstraction:

> *In addition to iteration and recursion, there are more forms of repetition used widely in programming but rarely studied in introductory programming. For instance, programmers in languages from Haskell to Python can use comprehensions . . . Big-data programmers use abstractions such as MapReduce . . . The users of these abstractions do not perform explicit iteration; that is hidden inside the implementation . . . Students exposed to SQL-style interfaces can possibly begin to program repetition—and hence tackle interesting data sets—quickly and with a much higher-level notional machine that does more behind the scenes* [83].

6.2.8 Expressibility. Some RPBs are easier to translate into words or pictures or activities than others; we call this EXPRESSIBILITY. More specifically, we mean how feasible it is to construct a student-facing form of the RPBs that is suitable for the target audience.

Formal rule systems may be more ACCURATE than informal ones but can also be harder to translate into a beginner-friendly form. Some rulesets can be readily expressed using metaphors or analogies to common-sense concepts (although care must be taken to draw the learners' attention to where the analogies fail and to facilitate transfer from the everyday concepts to the abstract RPBs [26, 147]). As an example, the Boxer environment for end-user programming [34] features a model of computation that deliberately exploits people's everyday spatial understandings; in this model, concepts such as scope are readily expressible using spatial metaphors.

6.2.9 Notational Fit. Although RPBs focus on behavior rather than syntax, their relationship with syntax influences their quality. Students "construct mental models of computation that are inconsistent with the actual behavior [in part because] the mapping from the surface syntax to intended behavior may not be obvious or may have multiple reasonable interpretations" [83].

Expressing and teaching RPBs is easier if there is a close and obvious mapping from the programming notation's surface concepts to the RPBs and the execution-time world of program behavior. We call this NOTATIONAL FIT. (N.B. This criterion is not limited to syntax alone but applies more generally to the notation's tangible parts and involves, for example, the names of library functions.)

NOTATIONAL FIT is reduced by additional concepts that are needed to explain program behavior, beyond those that are readily apparent in code. For example, our ruleset in Figure 8 uses the additional concepts of call stack and frame, which are not explicit in Python code and whose relationship to code requires clarification. The explanatory power gained from additional concepts needs to be weighed against the loss of notational fit.

Another consideration is the cardinality of the mapping from syntax to RPBs. A one-to-one mapping is uncomplicated: There is precisely one syntax for each behavior, and each keyword or other syntactical element is reserved for expressing a particular behavior. However, in many programming languages, the mapping is considerably more complicated, with multiple ways to express the same behavior and overloaded keyword semantics (cf. CONSISTENCY above).

NOTATIONAL FIT is better if runtime behavior is treated in terms of the notation itself. For example, in the substitution model for programs with immutable state [144], program runs are treated as transformations of the code, which replace each evaluated expression by its value.

Some attempts at improving NOTATIONAL FIT may reduce ACCURACY and SENSITIVITY TO CONCEPTIONS. For example, an RPB might say that "assignment statements replace what's on the

left-hand side with what's on the right." This expresses behavior in terms of the textual notation that learners use but fails to distinguish between the fundamentally different fragments of code that appear on the left and right. A ruleset that introduces memory storage as separate from code (as our example rulesets do) has poorer notational fit but may help students with the asymmetry of an assignment.

An example of high NOTATIONAL FIT can be found in **programming-by-example (PbE)** systems. In PbE, the programmer demonstrates what they would like the program to do by directly manipulating a graphical representation of the problem domain, and the system constructs the program from these interactions [87]. PbE's notational fit comes at the cost of GENERALITY and TRANSFERABILITY [116, 130]. A similar unity of notation and behavior is achieved in Hauswirth and colleagues' educational languages "that *are* the system, instead of languages that are *about* the system" [67]. For example, one of these languages is programmed by placing physical toy railroads according to certain rules; the program is run by making a toy engine run along the tracks—that is, within the program itself. The RPBs for such a language can be phrased in terms of the concrete manipulatives that the "code" consists of. The Boxer environment [34] implements a related approach, following a principle of "naive realism" so that its programmers can "pretend that what they see on the screen is their computational world in its entirety." For example, Boxer's variables are visible and directly manipulable in the same display where code can be typed and run.

6.2.10 Generality. Many models of program behavior are, by design, specific to a language. Some RPBs are particularly specific: rulesets for domain-specific languages have limited GENERALITY, as do RPBs that explain microworlds in terms of domain concepts.

It is, however, possible to write rulesets that seek GENERALITY over multiple sufficiently similar languages. Such an aspiration is embodied in some program-visualization tools, which have been intended as "language-independent" or at least visualize the same model of program behavior for several languages [74, 136, 138]. The increase in generality comes with a loss of ACCURACY and/or COVERAGE, as only an intersection of the languages can be covered accurately. Pollock et al. [107] describe a more nuanced approach to creating visualizations that are "consistent across languages" by generating them automatically from formal semantics.

Even if an entire ruleset does not generalize, partial generality can be assessed by considering which individual rules apply to many languages.

6.2.11 Transferability. In many cases, TRANSFERABILITY is a more realistic goal than GENERALITY. The difference is that whereas GENERALITY is about how broadly the rules apply, TRANSFERABILITY is about how manageable the transfer gap is from the rules to other languages or rulesets that the learners are expected to encounter—that later learning may happen at a specific point of a multilingual progression or at some other time in the learners' foreseeable future.

Transfer between programming languages does take place but tends to require explicit guidance and is known to be challenging for novices and even seasoned programmers (see, e.g., References [8, 51, 57, 122, 125, 143]). Santos et al. [122] (p. 40) write on learning a sequence of languages:

> *Knowledge of prior languages [matters], because the concepts in different languages are often closely related but different in important ways. We believe that it is thus important . . . to explicitly bridge between subsequent languages. Part of this bridge can be built in the initial course, by establishing bridgeheads when key concepts are introduced. The subsequent course can then build bridges that connect to the corresponding bridgeheads.*

To improve TRANSFERABILITY, a multilingual RPB progression can identify connections and disconnections between successive models of program behavior. Rulesets can be annotated to point up needs and opportunities for bridge-building, guided transfer, and communication

between instructors. Such designs can be informed by research on the specific difficulties in conceptual transfer between languages or paradigms (e.g., References [122, 143]).

Transferability is a potential downside of domain-specific RPBs, as special attention may be needed to mediate transfer to general-purpose languages. Dann et al. [28] noted the need for support in transfer from the visual behaviors of the Alice environment to Java programming, and Touretzky et al. [142] comment on the challenge of transfer from the Kodu microworld to "procedural languages such as Scratch or Python." However, Touretzky et al. note that a general appreciation for the rule-based "lawfulness" of program behavior—which can be learned through Kodu—is a valuable learning goal in itself [141]; moreover, "mental simulation and analytical reasoning are important in all types of programming. If children become concrete operational thinkers about Kodu programs, then we expect they will then be quicker to reach this developmental stage when learning other languages" [142].

Of course, in addition to a model of program behavior, other things also matter for transfer, such as syntax and problem-solving patterns. As ever, RPBs can cover only part of the picture.

6.2.12 Sensitivity to Conceptions. This criterion is concerned with how RPBs attend to the learners' expected prior knowledge, common (mis)conceptions, and difficult-to-learn content.

Sensitivity to conceptions can be direct or indirect. The former means singling out selected conceptions in the RPB text, such as in our asterisked RPBs that contrast with relevant non-normative conceptions. The latter means otherwise designing the model of program behavior so that it can help students overcome known conceptual difficulties.

Teachers and researchers have identified many trouble spots in learning to program, such as arguments vs. parameters, printing vs. returning, assignment statements, and object instantiation [36, 114, 133]. Ideally, an RPB design is sensitive to these known difficulties. Sufficient granularity is needed to express the critical features of the challenging phenomena, and designers need to watch out for excessive attempts at simplicity that blur key distinctions.

Sensitivity to conceptions can look very different when considered for an entire progression rather than a single ruleset. A design choice that has some immediate benefits may encourage a limited or mistaken conception that bites back sooner or later, as the learners progress.

A possible design trap is to introduce every new concept with the simplest imaginable code and RPBs to suit. For example, when teaching parameter passing, some teachers initially use only atomic argument expressions—i.e., simple literals and variable variables—as in `myFunc(1, 3)` or `myFunc(myVar)`. Such code can be explained in terms of simple RPBs that do not cover the general case. However, the relationship between arguments and parameters is a notoriously tricky concept for beginners, with many misconceptions and general confusion reported in the literature (e.g., References [53, 79]). For instance, students sometimes perceive parameter passing as the creation of links between variables in the calling code and the function. The simplest code examples and RPBs are (we suggest) not ideal for addressing these issues. Our example rulesets treat expressions as a general concept early and build on that in the later RPBs that add parameter passing.

Similar design concerns apply to many other concepts that learners may initially encounter only in a simple form or particular use case, which limits how they perceive the new concept and link it to other concepts (e.g., Boolean expressions may be perceived as a special syntax in `if` statements rather than expressions that are evaluated as usual and usable in a variety of ways). As we noted in Section 4.2, students infer their own "rules" about programming; those rules are often too restrictive, which can be addressed by having students practice on a rich variety of programs [134] and by teaching RPBs that link those programs together with general principles.

There is also the possibility that an RPB design might do harm: are there difficulties that might be triggered or exacerbated by the RPBs? For example, learners are known often to conflate objects with the referring variables and think of the variable names as properties of the objects [72, 131],

which causes problems with aliasing and scope. An RPB that treats assignment as "naming" an object (or list) does distinguish imperative assignment from mathematical equality (thus partially attending to one common misconception); however, it could encourage variable–object conflation unless the relationship between names and objects is carefully taught.

RPB designers must consider what is not highlighted by the system but nevertheless needs to be learned. For example, Grover and Basu [62] comment on how Scratch de-emphasizes types: *"[students who] don't have to deal with variables' data types . . . end up with an incomplete understanding of expressions and operators (e.g. the fact that arithmetic operators make sense only with numbers [is just one] of the many issues)."* Franklin et al. [57] discuss variable initialization in Scratch similarly. RPBs that deal with non-obvious aspects of the system should be constructed with particular care.

6.2.13 Wieldiness. This criterion is about how convenient RPBs are to use. It may be considered separately for different tasks, but there is one task that stands out: tracing code. Some RPBs are clunkier than others as tracing tool.

Wieldiness can be at odds with high granularity: more detail means more working-memory load and more work to keep track of a program run. Mutable state is another consideration: Krishnamurthi and Fisler [83] suspect that although mutability can make program-writing quicker, it may greatly increase the cognitive load of tracking state. In contrast, some of the techniques that improve notational fit can also boost wieldiness. If program behavior is viewed as transformations of, or events within, then the notation itself, the need for a separate "status representation" [104] in memory is reduced, as the notational context supports *distributed cognition* [71].

Even a simple ruleset with high notational fit is not necessarily wieldy. Tunnell Wilson et al. [144] report that a weakness in the substitution model of evaluating immutable-state programs is that it can be laborious to use, especially when dealing with large data structures. That unwieldiness, the authors suggest, may predispose learners to take unsafe "shortcuts" when tracing.

The presence of static typing may influence how easy it is to apply RPBs for tracing. A colleague recently remarked to us: "My [mental model of a] notional machine runs faster if I can leave type checking to the compiler." (If there is research investigating this, then we are not aware of it.)

6.2.14 Assessability. Knowledge of any model of program behavior can be assessed indirectly by teaching the model and then observing how successful learners are at tracing or some other task. A ruleset is more assessable if it lends itself to a more detailed analysis of which rules the learners have or have not successfully applied.

Nelson et al. [96] observe that typical programming assessments "have items that require 10–20 concepts to answer correctly, even for basic skills like program tracing," which is a problem for formative assessment especially (see also Reference [91, 92]). As an improvement, Nelson et al. [96] seek assessments of program-tracing knowledge that can target individual constructs separately and in specific combinations. For an RPB ruleset to support such assessment, its rules and concepts need to be sufficiently detailed and as distinct from each other as possible.

6.2.15 Implementability. A limited form of implementability is that a human can, with reasonable effort, write the rules into a program such as a semantics game [4] or a program visualization.

A stronger form of implementability is machine-readability: A computer can read in the ruleset and operate on it. For instance, program visualization might be automatically created from formal rulesets [107]. The generation of educational examples is another prospect (cf. References [97, 112]).

6.2.16 Clarity of Writing. Our other evaluation criteria are tied to the RPBs' content: They reflect what the model of program behavior is like. This last criterion is about the way the RPBs are expressed as text. Essentially, it asks whether the model of program behavior is communicated well to the RPBs' target audience of teachers and/or researchers.

7 DISCUSSION

Although few teachers, textbooks, tools, and curricula explicitly describe a model of program behavior, many nevertheless implicitly target one. Fewer still set down detailed progressions of models in writing. It is often difficult to disentangle learning goals from visualizations and other teaching methods [32]. We believe the framework presented in this article can help.

A lightweight way to use the RPB framework is to adopt it as a tool for thinking and discussing, without actual, written-down RPBs. Even without RPBs as such, distinguishing between syntax, models of program behavior, and teaching methods may crystallize instructional-design decisions and research questions. Mentally or textually sketching out RPBs can help educators define what they hope or expect students to learn and consider whether a method matches those goals.

Full use of the framework requires some more effort: The RPBs need to be designed and written. However, not everyone has to write their own from scratch, which is as well given that documenting detailed learning objectives is not most teachers' favorite activity. RPBs can be shared and published. Teachers and researchers can adopt and adapt RPBs from others.

7.1 Using the RPB Framework to Look at Notional Machines

Our work derives from, parallels, and extends the literature on notional machines. We feel that the breakdown between models of program behavior, written-down RPBs, and teaching methods (Figure 4 on p. 6 above) has helped us debate notional machines and related concepts with greater precision. We hope that the framework will help people to interpret the many meanings of "notional machine" in the literature and to understand each other.

Our design advice, evaluation criteria, and example progression (Sections 5 and 6) are potentially useful for designing notional machines even without otherwise adopting the RPB framework.

7.2 Using RPBs in Instructional Design

At the very beginning of this article, Amal wanted to know what exactly they should teach about program behavior. Dana wanted to know how to pick a teaching method. The RPB framework could help teachers like Amal to establish what the target is and teachers like Dana to analyze how existing methods match their specific, contextual needs. As the long and yet incomplete review in this article's web supplement [40] illustrates, there is a bewildering and growing array of tools and methods whose suitability depends in part on the chosen model of program behavior. Once the target RPBs are established, the teacher can use them as a guide and a checklist when selecting or creating a suitable combination of activities, visualizations, metaphors, and other tools. RPB progressions can assist in planning learning trajectories and minimizing negative transfer.

Many instructors would like to have students first read and/or write simpler programs before gradually advancing to increasingly complex programs. The instructor thus needs to order programs by cognitive complexity, which is non-trivial. To address this, Duran et al. [39] have proposed an extension to cognitive schema theory, characterizing each program's complexity in terms of the hierarchical schemas that the programmer uses to reason about the program. The lowest-level schemas correspond to the mechanical operations of a model of program behavior, meaning that a program's cognitive complexity is contingent on the model that students learn. RPB rulesets and progressions could thus provide a foundation for cognitive analyses of program complexity and inform task sequencing in instructional design.

Neo-Piagetian theories of learning posit that learners progress through certain stages of development that reoccur within each (sub)domain of learning; different stages call for different learning activities. Working from this perspective, Lister [88] describes how learners progress from a *sensorimotor* stage of not being able to trace code to a *pre-operational* stage (and beyond). This pattern

repeats after new concepts cause the learner's earlier abilities to break down, triggering *accommodation* and a return to an earlier developmental stage. We see RPB progressions as a way to mark where a new ruleset may disrupt an earlier understanding of program behavior and necessitate a new cycle through the stages.

RPB progressions may assist teachers in designing scaffolding that supports learners as they progress toward richer understandings of program behavior. In master–apprentice pedagogies such as cognitive apprenticeship [68], the master might use RPBs to record aspects of their tacit knowledge and to identify candidate zones of proximal development for exploring together with the apprentices.

7.3 Using RPBs to Document Teaching and Research

Our imaginary teacher Bao wanted to decide which parts of their course design to document for others' benefit. Perhaps Bao means to share the design with local colleagues, such as teaching assistants or teachers of related courses. RPBs can be part of such a description. Teachers might document RPBs for personal purposes as well, and learn as they do so. Documenting pitfalls and potential misconceptions into an RPB ruleset further enriches these descriptions.

Dissemination of practice to other contexts is another reason for sharing. A teacher edition of a textbook could list RPBs. A training course might ask pre-service teachers to critique a course's RPB progression (or design and discuss new RPBs). RPBs could be collected in a repository.

Concurrently with our work for this article, Fincher et al. [47] have been working toward a catalogue that helps teachers select and compare notional machines (which Fincher et al. define as student-facing "pedagogic devices"; i.e., teaching methods). The models of program behavior behind these teaching methods could be recorded as RPBs. Moreover, teachers who have a particular RPB ruleset in their sights might use the catalogue to guide their selection of teaching methods. Overall, we see the present work and that of Fincher et al. [47] as complementary.

Researchers like Camille from our Prelude need to compare assessment results from different programming courses. The easy assumption is that two courses have similar goals if they use the same programming language and cover the same constructs. The reality is much more nuanced, as students may learn very different styles, patterns, tools—and models of program behavior.

We call on empirical researchers not only to report programming languages but also to reference the RPBs that the learners had been taught. To draw incisive conclusions, just knowing "the students had been taught variables, functions, and loops" is often insufficient, as it would help to know *what* the students had been taught about those constructs and how the students were expected to reason about code that features them. Multi-institutional studies and meta-analyses compare different cohorts of learners assessed at different stages of different programming courses; RPB progressions could help to make such comparisons fairer, or at least to illustrate their shortcomings.

We also call on the creators of educational visualizations to set down the RPBs that the visualizations are designed for (even though the visualization might work with other RPBs as well [32]).

In this article, we have discussed the use case where a single instructional designer deliberately identifies RPBs and teaching methods. However, RPBs might also find a different use as a tool for analyzing instructional designs created without RPBs, post hoc. For example, researchers might extract rulesets and progressions from a textbook by extrapolating from the student-facing materials, with or without the book author's assistance.

7.4 Using RPBs to Sharpen Research Questions

The RPB framework suggests new research questions and new, more precise forms of old ones.

Some obvious questions to ask are as follows: Which models of program behavior suit which contexts? What are the benefits and drawbacks of adopting a particular RPB ruleset? What makes a

good RPB progression? When should students learn multiple perspectives—multiple rulesets—that provide complementary perspectives to program behavior? Our design criteria from Section 6.2 provide a vocabulary for exploring such questions. To date, studies that directly compare models of program behavior—or notional machines in this sense—are rare (but do exist [144]).

Another broad and important question is how to teach a particular model. Which methods work for which RPBs and how does that depend on context? Conversely, which RPBs is a particular method—a visualization, say—suited for? Are there methods that work well for an entire RPB progression or many alternative rulesets?

The thread of CER that explores students' conceptions and misconceptions of programming concepts is alive and well. RPBs provide a normative target model against which learners' conceptions may be viewed. Studies could examine how the choice of RPBs and/or teaching methods affects the frequency of (mis)conceptions. In some cases, having a course's learning objectives detailed as RPBs should help researchers (and teachers) identify why a particular conception is frequent.

The model of program behavior that learners are exposed to affects what they can learn. The validity of, say, a program-tracing assessment depends on the target ruleset as well as the programming language [83]. This means that great care is needed when designing concept inventories for multiple languages (e.g., References [17, 99]) or interpreting findings from such instruments. RPBs could support the design of assessments, whether formative or summative. For example, assessment questions might target specific RPBs or RPB combinations.

Computing educators are locked in an eternal struggle among themselves about which languages to teach. RPBs do not resolve this conflict but they could inform some of the discussions: the choice of language and paradigm should be influenced by which models of program behavior one wishes to teach. Researchers might explore which rulesets are suited to which languages, or which language—with its syntax, libraries, and so on—is the best vehicle for teaching a particular ruleset (or a language-specific variant of a generic ruleset). Notional machines already feature occasionally in language and paradigm debates; see Section 6.2.4 above for examples.

Although initiatives in programming education have tended to focus on a single language, software and other tools have been developed for cross-language transfer as well. Krishnamurthi and Fisler [83] point to many open research questions about transfer and argue that "the similarity between two languages is the extent to which a notional machine for one gives an accurate account of the behavior of the other." Tshukudu and Cutts [143] are exploring the differences in how different concepts transfer between languages. The topic of transfer is sizzling hot in primary school education, with an explosion of recent research on the transition from blocks-based to text-based environments. RPBs can record the similarities and differences between stages and contribute to the design of these tools and studies. As an increasingly diverse worldwide population of programmers uses an ever greater number of general-purpose and domain-specific (and even task-specific [65]) programming languages, a future seems likely that requires people to understand many languages and other programming notations [3] and to fluently switch between them [66]. If that happens, then multilingual RPB progressions will be more useful still.

The relationship between natural and programming languages is another trendy topic. Researchers track eyes and scan brains to explore whether and how programming resembles natural-language use, with a clear picture yet to emerge [55, 75, 84, 103, 111, 127]. The languageness of programming notations is being both celebrated and questioned as restrictive [3, 15, 56]. Ideas from language learning and literacy education are working their way into programming pedagogy [10, 15, 56, 70, 100, 108, 143]. Natural languages, like programming languages, have rules; people acquire the rules of their native language implicitly and may do so even for subsequent languages. Should it turn out that the cognitive similarities between programming and natural languages are substantial, the impact on programming education would be great.

A target semantics set down as RPBs could help researchers to explore the implicit acquisition of programming rules, to compare the explicit teaching of rules to implicit acquisition through constant exposure, or to evaluate other pedagogies inspired by language learning.

Practice is widely cited as crucial to the development of students' code-tracing skills (and by extension their code-writing skills), but there is a variety of views on which type of practice is ideal. The PLTutor system [97] is a recent example of having students practice semantic rules on *decontextualized* code fragments. In contrast, Lowe [90] proposes that rather than targeting rules directly, beginners should practice contextualized and scaffolded debugging activities, which motivate tracing and are more likely to generate intuitive, automatic mental representations that transfer. A combination of different practice tasks is another possibility, of course. Again, explicit RPBs could provide a foundation for studies that compare these approaches.

Program-comprehension frameworks such as the Block Model [124] highlight the diversity of knowledge needed in programming. RPBs reflect just one kind; others include syntactic knowledge, domain knowledge, and solution schemas at various levels of abstraction. Danielak [27] urges researchers to recognize that learners employ a variety of models as they deal with programs and make many moment-to-moment decisions about which knowledge to draw on. These models are not limited to the mechanistic behavior captured in our RPBs and include, for example, domain-specific metaphors and ways of reasoning. Further research is needed to explain how RPB-like knowledge of program behavior interacts with other programming and domain knowledge, and which contextual factors prompt learners to use different knowledge as they read and write programs.

RPBs record teacher-set goals, but a similar approach might also be applied to student knowledge. Researchers might elicit students' conceptions of program behavior and write them down as "rules" of some description. Whether it makes sense to treat students' conceptions as relatively stable "rulesets" or incoherent collections of separate context-bound elements is an open question being debated by conceptual-change theorists (Section 4.2). Either way, documenting more student-constructed rules would help CER characterize how students' reason about programs and trace them mentally. Lewis et al. [86] discuss early work by Davis et al. [29] in this vein.

Our example RPBs apply to the sort of text- or block-based programs that tend to appear in introductory programming courses. Other forms of programming could also be expressed as RPBs, including machine learning (cf. Reference [46]), embedded systems (cf. Reference [23]), visual programming and direct-manipulation interfaces (cf. Section 6.2.9), and so on.

In this article, we have posited that RPBs are written by instructional designers. Future research might explore how to guide *students* to identify and write down rules that explain program behavior.

7.5 Limitations

While we have ourselves found the RPB framework helpful for organizing our thoughts about notional machines and related phenomena, we do not have evidence of how understandable or usable the framework is by other researchers and practitioners.

We have not detailed a *process* for designing RPBs or evaluating designs. We have identified evaluation criteria, but we do not expect our list to be final. On the contrary, we hope that others might refine and extend the list.

We have not presented empirical evidence for or against any particular RPB ruleset or progression, and indeed it is not the purpose of this article to attempt such an argument.

Although the basic notion of RPBs is not inextricably tied to any specific epistemology or theory of knowledge (such as mental models or a particular theory of conceptual change), the framework does rely on some fundamental assumptions and this article does lean toward certain theoretical

perspectives. Much of the research that we have cited in support of the RPB framework and the evaluation criteria derives from cognitive psychology. Moreover, our discussion in this article assumes that targeting a certain kind of understanding is a pragmatic thing to do in education (which is not to say that RPBs attempt to tell an objective "truth"; on the contrary, we encourage targeting different kinds of models as befits each context). We acknowledge that this assumption is not universally accepted and that there are tensions between the RPB framework as presented here and other perspectives on program behavior. For example, some social-constructivist theories posit that the teachers should not attempt to set particular knowledge as a target but should instead explore different ways of meaning-making together with learners. A related concern is that a particular teacher-defined model of program behavior might not suit all learners within a context equally well, so targeting a single model might hinder the learning of some students; this is a caveat in applying the RPB framework.

Overall, more research is needed to explore whether and how RPBs may fit with different epistemologies and theories of learning. One plausible use for an RPB ruleset could be to repurpose the RPBs as a point of reference while exploring the viable and non-viable aspects of learners' various conceptions, rather than treating the RPBs as a normative target.

8 CONCLUSION

We join others in bringing models of program behavior into the spotlight as explicit learning objectives. To that end, we have proposed a framework for setting down these models as teacher-facing rules, rulesets, and progressions of rulesets. We have argued that the RPB framework provides structure to ongoing research and discussions around notional machines and programming-language semantics in education; the framework might also help these ideas gain greater purchase among practitioners. We have outlined how RPBs may assist in instructional design, help document teaching and research, and sharpen research questions and tool designs. We have nominated a set of evaluation criteria for models of program behavior and the RPBs that describe those models; we have discussed tradeoffs between those criteria and presented an example progression of RPBs as a proof-of-concept example.

We encourage instructional designers and researchers to describe models of program behavior, whether in terms of the RPB framework or otherwise. We invite the computing education community to formulate research questions that distinguish between models of program behavior and methods for teaching those models.

This article is part of a burst of scholarly activity around program behavior, which is being looked at through the lenses of semantics [83], visualizations [32], and pedagogical practices [48]. We hope that our contribution will help to connect these perspectives and, ultimately, contribute toward better programming education.

ACKNOWLEDGMENTS AND NOTE

We thank the participants of Dagstuhl Seminar 19281 [64] for discussions that inspired this work.

Some of the ideas presented here appear as part of the first author's doctoral dissertation [38]. This article greatly extends that incipient text and has been thoroughly rewritten.

REFERENCES

[1] Sarah Alhammad, Shirley Atkinson, and Liz Stuart. 2016. The role of visualisation in the study of computer programming. In *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group (PPIG'16)*.

[2] Aivar Annamaa. 2015. Introducing Thonny, a Python IDE for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling'15)*. ACM, 117–121.

[3] Ian Arawjo. 2020. To write code: The cultural fabrication of programming notation and practice. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'20)*. ACM, 1–15.

[4] Ian Arawjo, Cheng-Yao Wang, Andrew C. Myers, Erik Andersen, and François Guimbretière. 2017. Teaching programming with gamified semantics. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'17)*. ACM, 4911–4923.

[5] Ferdinando Arzarello, Giampaulo P. Chiappini, Enrica Lemut, Nicolina Malara, and Michele Pellerey. 1993. Learning programming as a cognitive apprenticeship through conflicts. In *Cognitive Models and Intelligent Environments for Learning Programming*, Enrica Lemut, Benedict du Boulay, and Giuliana Dettori (Eds.). Springer, 284–298.

[6] John William Atwood and Eric Reneger. 1981. Teaching subsets of pascal. *SIGCSE Bull.* 13, 1 (1981), 96–103.

[7] Mohammad Reza Azadmanesh and Matthias Hauswirth. 2017. Concept-driven generation of intuitive explanations of program execution for a visual tutor. In *Proceedings of the IEEE Working Conf. on Software Visualization (VISSOFT'17)*. IEEE.

[8] Gina R. Bai, Joshua Kayani, and Kathryn T. Stolee. 2020. How graduate computing students search when using an unfamiliar programming language. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC'20)*. ACM, 160–171.

[9] T. Balman. 1981. Computer assisted teaching of FORTRAN. *Comput. Educ.* 5, 2 (1981), 111–123.

[10] Brett A. Becker. 2019. Parlez-vous Java? Bonjour La Monde != Hello World: Barriers to programming language acquisition for non-native English speakers. In *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group (PPIG'19)*.

[11] Mordechai Ben-Ari. 2001. Constructivism in computer science education. *J. Comput. Math. Sci. Teach.* 20, 1 (2001), 45–73.

[12] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. 2011. A decade of research and development on program animation: The Jeliot experience. *J. Vis. Lang. Comput.* 22, 5 (2011), 375–384.

[13] Dave Berry. 1990. *Generating Program Animators from Programming Language Semantics*. Ph.D. Dissertation. University of Edinburgh.

[14] Michael Berry and Michael Kölling. 2014. The state of play: A notional machine for learning programming. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE'14)*. ACM Press, New York, NY, 21–26.

[15] Marina Umaschi Bers. 2019. Coding as another language: A pedagogical approach for teaching computer science in early childhood. *J. Comput. Educ.* 6, 4 (2019), 499–528.

[16] Andrew P. Black and Kim B. Bruce. 2018. Teaching programming with grace at portland state. *J. Comput. Small Coll.* 34, 1 (2018), 223–230.

[17] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and validating Java misconceptions toward a CS1 concept inventory. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'19)*. ACM, New York, NY, 23–29.

[18] Michael E. Caspersen and Jens Bennedsen. 2007. Instructional design of a programming course — A learning theoretic approach. In *Proceedings of the 3rd International Computing Education Research Workshop (ICER'07)*. ACM, 111–122.

[19] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually learning programming supported by a growable programming language. *IEEE Trans. Emerg. Top. Comput.* 4, 3 (2016), 404–415.

[20] Jie Chao, David F. Feldon, and James P. Cohoon. 2018. Dynamic mental model construction: A knowledge in pieces-based explanation for computing students' erratic performance on recursion. *J. Learn. Sci.* 27, 3 (2018), 1–43.

[21] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an algebraic stepper. In *Lecture Notes in Computer Science*. Vol. 2028. Springer, 320–334.

[22] Frank Coffield and Sheila Edward. 2009. Rolling out 'good', 'best' and 'excellent' practice. What next? Perfect practice? *Br. Educ. Res. J.* 35, 3 (2009), 371–390.

[23] Bill Collis. 2014. *A Visualiser for Embedded Systems—Development of a Visualiser to Support Novice Learners' Understandings of Embedded Systems*. Master's Thesis. The University of Auckland.

[24] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. In *Proceedings of the 13th International Computing Education ResearchWorkshop (ICER'17)*. ACM, New York, NY, 164–172.

[25] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice rationales for sketching and tracing, and how they try to avoid it. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'19)*. ACM, New York, NY, 37–43.

[26] Paul Curzon, Peter W. McOwan, James Donohue, Seymour Wright, and William Marsh. 2018. Teaching of concepts. In *Computer Science Education: Perspectives on Teaching and Learning in School*, Sue Sentance, Erik Barendsen, and Carsten Schulte (Eds.). Bloomsbury.

[27] Brian A. Danielak. 2019. Deprecating misconceptions through context-dependent accounts of productive knowledge. In *Proceedings of the 15th International Computing Education ResearchWorkshop (ICER'19)*. ACM, 91–100.

[28] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper. 2012. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. ACM Press, New York, NY, 141–146.

[29] Elizabeth A. Davis, Marcia C. Linn, Lydia M. Mann, and Michael J. Clancy. 1993. "Mind Your P's and Q's": Using parentheses and quotes in LISP. In *Proceedings of the 5th Workshop on Empirical Studies of Programmers*, Curtis R. Cook, Jean Scholtz, and James C. Spohrer (Eds.). Ablex Publishing, 62–85.

[30] Johan de Kleer and John Seely Brown. 1981. Mental models of physical mechanisms and their acquisition. In *Cognitive Skills and Their Acquisition*, John R. Anderson (Ed.). Lawrence Erlbaum, 285–309.

[31] Johan de Kleer and John Seely Brown. 1983. Assumptions and ambiguities in mechanistic mental models. In *Mental Models*, Dedre Gentner and Albert L. Stevens (Eds.). Lawrence Erlbaum, 155–190.

[32] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage against the machine: Rise of the notional machines as effective pedagogical devices. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'20)*. ACM, 159–165.

[33] Andrea A. DiSessa. 2018. A friendly introduction to "Knowledge in Pieces": Modeling types of knowledge and their roles in learning. In *Invited Lectures from the 13th International Congress on Mathematical Education*, Gabriele Kaiser, Helen Forgasz, Mellony Graven, Alain Kuzniak, Elaine Simmt, and Binyan Xu (Eds.). Springer, 65–84.

[34] Andrea A. DiSessa and Harold Abelson. 1986. Boxer: A reconstructible computational medium. *Commun. ACM* 29, 9 (1986), 859–868.

[35] Peter Donaldson and Quintin Cutts. 2018. Flexible low-cost activities to develop novice code comprehension skills in schools. In *Proceedings of the 13th Workshop in Primary and Secondary Computing Education (WiPSCE'18)*. ACM Press, New York, NY, 1–4.

[36] Benedict du Boulay. 1986. Some difficulties of learning to program. *J. Educ. Comput. Res.* 2, 1 (1986), 57–73.

[37] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: Presenting computing concepts to novices. *Int. J. Man–Mach. Stud.* 14 (1981), 237–249.

[38] Rodrigo Duran. 2020. *Cognitive Complexity of Comprehending Computer Programs*. Ph.D. Dissertation. Aalto University.

[39] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 14th International Computing Education ResearchWorkshop (ICER'18)*. ACM, New York, NY, 21–30.

[40] Rodrigo Duran, Juha Sorva, and Otto Seppälä. 2021. *Web Appendix for Rules of Program Behavior: A Review of Related Teaching Methods*. ACM. https://osf.io/r8kuq/?view_only=7e87778e40544567972b2a0256f4c86d.

[41] Anna Eckerdal and Michael Thuné. 2005. Novice Java programmers' conceptions of "object" and "class" and variation theory. *SIGCSE Bull.* 37, 3 (2005), 89–93.

[42] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference (ACE'18)*. ACM, 83–89.

[43] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The structure and interpretation of the computer science curriculum. *J. Funct. Program.* 14, 4 (2004), 365–378.

[44] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2014. *How to Design Programs* (2nd ed.). MIT Press.

[45] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket manifesto. In *Proceedings of the 1st Summit on Advances in Programming Languages (SNAPL'15)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl, 113–128.

[46] Rebecca Fiebrink. 2019. Machine learning education for artists, musicians, and other creative practitioners. *ACM Trans. Comput. Educ.* 19, 4 (2019), 1–32.

[47] Sally Fincher, Johan Jeuring, Craig Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional machines in computing education: The education of attention. In *ITiCSE Working Group Reports (ITiCSE-WGR'20)*. ACM, 21–50.

[48] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict Boulay, Matthias Hauswirth, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Capturing and characterising notional machines. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'20)*. ACM, 502–503.

[49] Robert Bruce Findler. 2020. DrRacket: The Racket Programming Environment. Retrieved July 29, 2020 from https://docs.racket-lang.org/drracket/index.html.

[50] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A programming environment for Scheme. *J. Funct. Program.* 12, 2 (2002), 159–182.

[51] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates. In *Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE'17)*. ACM, 213–218.

[52] Sue Fitzgerald, Beth Simon, and Lynda Thomas. 2005. Strategies that students use to trace code: An analysis based in grounded theory. In *Proceedings of the 1st International Workshop on Computing Education Research (ICER'05)*. ACM, 69–80.

[53] Ann E. Fleury. 1991. Parameter passing: The rules the students construct. *SIGCSE Bull.* 23, 1 (1991), 283–286.

[54] Ann E. Fleury. 2000. Programming in Java: Student-constructed rules. *SIGCSE Bull.* 32, 1 (2000), 197–201.

[55] Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE/ACM.

[56] Stephen R. Foster and Lindsay D. Handley. 2020. *Don't Teach Coding (Before You Read This Book)*. John Wiley & Sons.

[57] Diana Franklin, Charlotte Hill, Hilary A. Dwyer, Alexandria K. Hansen, Ashley Iveland, and Danielle Harlow. 2016. Initialization in Scratch: Seeking knowledge transfer. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, 217–222.

[58] Diana Franklin, Jean Salac, Cathy Thomas, Zene Sekou, and Sue Krause. 2020. Eliciting student Scratch script understandings via Scratch Charades. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. ACM, 780–786.

[59] Kathryn E. Gray and Matthew Flatt. 2003. ProfessorJ: A gradual introduction to Java through language levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. ACM Press, New York, NY, 170–177.

[60] Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: A 'Cognitive Dimensions' Framework. *J. Vis. Lang. Comput.* 7, 2 (1996), 131–174.

[61] Thomas L. Griffiths and Joshua B. Tenenbaum. 2009. Theory-based causal induction. *Psychol. Rev.* 116, 4 (2009), 661–716.

[62] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming. In *Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE'17)*. ACM, New York, NY, 267–272.

[63] Philip J. Guo. 2018. Non-native english speakers learning computer programming: Barriers, desires, and design opportunities. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'18)*. ACM Press, New York, NY, 1–14.

[64] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. *Report from Dagstuhl Seminar 19281: Notional Machines and Programming Language Semantics in Education*. Technical Report 7. Schloss Dagstuhl–LZI GmbH. Retrieved July 29, 2020 from https://www.dagstuhl.de/19281.

[65] Mark Guzdial and Bahare Naimipour. 2019. Task-specific programming languages for promoting computing integration: A precalculus example. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli Calling'19)*. ACM, New York, NY, 1–5.

[66] Rebecca L. Hao and Elena L. Glassman. 2020. Approaching polyglot programming: What can we learn from bilingualism studies? In *Proceedings of the 10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'19)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 1–7.

[67] Matthias Hauswirth, Andrea Adamoli, and Mohammad Reza Azadmanesh. 2017. The program is the system: Introduction to programming without abstraction. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research (Koli Calling'17)*. ACM Press, New York, NY, 138–142.

[68] Sara Hennessy. 1993. Situated cognition and cognitive apprenticeship: Implications for classroom learning. *Stud. Sci. Educ.* 22, 1 (1993), 1–41.

[69] Felienne Hermans. 2018. Code.org–Baker Franke. Retrieved July 29, 2020 from http://www.felienne.com/archives/5920.

[70] Felienne Hermans. 2020. Hedy: A gradual language for programming education. In *Proceedings of the 16th International Computing Education ResearchWorkshop (ICER'20)*. ACM.

[71] James Hollan, Edwin Hutchins, and David Kirsh. 2000. Distributed cognition: Toward a new foundation for human–computer interaction research. *ACM Trans. Comput.-Hum. Interact.* 7, 2 (2000), 174–196.

[72] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding object misconceptions. *SIGCSE Bull.* 29, 1 (1997), 131–134.

[73] Richard C. Holt and David B. Wortman. 1974. A sequence of structured subsets of PL/I. In *Proceedings of the 4th ACM Technical Symposium on Computer Science Education (SIGCSE'74)*. ACM, 129–132.

[74] Johannes Holvitie, Teemu Rajala, Riku Haavisto, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. 2012. Breaking the programming language barrier: Using program visualizations to transfer programming knowledge in one programming language to another. In *Proceedings of the 12th IEEE International Conference on Advanced Learning Technologies (ICALT'12)*. IEEE, 116–120.

[75] Anna Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H. Kean, Riva Dhamala, Una-May O'Reilly, Marina U. Bers, and Evelina Fedorenko. 2020. Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife* 9 (2020), 1–24.

[76] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering program comprehension in novice programmers—Learning activities and learning trajectories. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'19)*. ACM, 27–52.

[77] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. 2020. Analysis of student misconceptions using Python as an introductory programming language. In *Proceedings of the 4th Conference on Computing Education Practice (CEP'20)*. ACM.

[78] James A. Juett. 2016. *Using Program Visualization to Illuminate the Notional Machine*. Ph.D. Dissertation. University of Michigan.

[79] Maria Kallia and Sue Sentance. 2019. Learning to use functions: The relationship between misconceptions and self-efficacy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, 752–758.

[80] Amy J. Ko. 2018. Designing and Evaluating Programming Languages: Dagstuhl Trip Report. Retrieved July 29, 2020 from https://medium.com/bits-and-behavior/designing-learnable-teachable-and-productive-programming-languages-dagstuhl-trip-report-81e41bde84bd.

[81] Tobias Kohn and Dennis Komm. 2018. Teaching programming and algorithmic complexity with tangible machines. In *Proceedings of the 11th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP'18)*. Springer, 68–83.

[82] Tobias Kohn and Bill Manaris. 2020. Tell me what's wrong: A Python IDE with error messages. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. ACM, New York, NY, 1054–1060.

[83] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming paradigms and beyond. In *Cambridge Handbook of Computing Education Research* (1st ed.), Sally Fincher and Anthony V. Robins (Eds.). Cambridge University Press, 377–413.

[84] Ryan Krueger, Tyler Santander, Westley Weimer, and Kevin Leach. 2020. Neurological divide: An fMRI study of prose and code writing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. IEEE/ACM.

[85] Colleen M. Lewis. 2012. *Applications of Out-of-Domain Knowledge in Students' Reasoning about Computer Program State*. Ph.D. Dissertation. UC Berkeley.

[86] Colleen M. Lewis, Michael Clancy, and Jan Vahrenhold. 2019. Student knowledge and misconceptions. In *Cambridge Handbook of Computing Education Research*, Sally Fincher and Anthony V. Robins (Eds.). Cambridge University Press.

[87] Henry Lieberman (Ed.). 2001. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann.

[88] Raymond Lister. 2016. Toward a developmental epistemology of computer programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education (WiPSCE'16)*. ACM, 5–16.

[89] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.* 36, 4 (2004), 119–150.

[90] Tony A. Lowe. 2019. Debugging: The key to unlocking the mind of a novice programmer? In *Proceedings of the IEEE Frontiers in Education Conference (FIE'19)*. IEEE.

[91] Andrew Luxton-reilly and Andrew Petersen. 2017. The compound nature of novice programming assessments. In *Proceedings of the 19th Australasian Computing Education Conference (ACE'17)*. ACM.

[92] Andrew Luxton-Reilly, Jacqueline Whalley, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, and Simon. 2017. Developing assessments to determine mastery of programming fundamentals. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'17)*. ACM, 47–69.

[93] Lauren E. Margulieux, Richard Catrambone, and Laura M. Schaeffer. 2018. Varying effects of subgoal labeled expository text in programming, chemistry, and statistics. *Instruct. Sci.* 46 (2018), 707–722.

[94] Craig S. Miller. 2014. Metonymy and reference-point errors in novice programming. *Comput. Sci. Educ.* 24, 2–3 (2014), 123–152.

[95] Mirjam Neelen and Paul A. Kirschner. 2019. Tackling Misconceptions Through Conceptual Change—Part II. Retrieved February 2, 2021 from https://3starlearningexperiences.wordpress.com/2019/07/10/tackling-misconceptions-through-conceptual-change-part-2/.

[96] Greg L. Nelson, Andrew Hu, Benjamin Xie, and Amy J. Ko. 2019. Towards validity for a formative assessment for language-specific program tracing skills. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli Calling'19)*. ACM.

[97] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 13th International Computing Education Research Workshop (ICER'17)*. ACM, New York, NY.

[98] Gökhan Özdemir and Douglas B. Clark. 2007. An overview of conceptual change theories. *Eurasia J. Math. Sci. Technol. Educ.* 3, 4 (2007), 351–361.

[99] Miranda C. Parker and Mark Guzdial. 2016. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proceedings of the 12th International Computing Education Research Conference (ICER'16)*. ACM, 93–101.

[100] Alex Parry. 2020. Investigating the relationship between programming and natural languages within the PRIMM framework. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education (WiPSCE'20)*. ACM.

[101] Richard E. Pattis. 1993. The "procedures early" approach in CS 1: A heresy. *SIGCSE Bull.* 25, 1 (1993), 122–126.

[102] Roy D. Pea. 1986. Language-independent conceptual 'bugs' in novice programming. *J. Educ. Comput. Res.* 2, 1 (1986), 25–36.

[103] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? An Eye tracking study. In *Proceedings of the 28th IEEE/ACM International Conference on Program Comprehension (ICPC'20)*. IEEE/ACM.

[104] David N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of learning in novice programmers. *J. Educ. Comput. Res.* 2, 1 (1986), 37–55.

[105] David N. Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Empirical Studies of Programmers*, Elliot Soloway and Sitharama Iyengar (Eds.). Ablex Publishing, 213–229.

[106] Jean Piaget. 1976. Piaget's theory. In *Piaget and His School*, Bärbel Zwingmann, Harold H. Inhelder, and Charles Chipman (Eds.). Springer, Berlin, 11–23.

[107] Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. 2019. Theia: Automatically generating correct program state visualizations. In *Proceedings of the ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E'19)*. ACM Press, New York, NY, 46–56.

[108] Scott R. Portnoff. 2018. The introductory computer programming course is first and foremost a LANGUAGE course. *ACM Inroads* 9, 2 (2018), 34–52.

[109] Patrice Potvin, Lucian Nenciovici, Guillaume Malenfant-Robichaud, François Thibault, Ousmane Sy, Mohamed Amine Mahhou, Alex Bernard, Geneviève Allaire-Duquette, Jérémie Blanchette Sarrasin, Lorie Marlène Brault Foisy, Nancy Brouillette, Audrey Anne St-Aubin, Patrick Charland, Steve Masson, Martin Riopel, Chin Chung Tsai, Michel Bélanger, and Pierre Chastenay. 2020. Models of conceptual change in science learning: Establishing an exhaustive inventory based on support given by articles published in major journals. *Stud. Sci. Educ.* 56, 2 (2020), 157–211. DOI: https://doi.org/10.1080/03057267.2020.1744796

[110] Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. 2007. Through the looking glass: Teaching CS0 with Alice. In *Proceedings of the 38th ACM Technical Symposium on Computer Science Education (SIGCSE'07)*. ACM, 213–217.

[111] Chantel S. Prat, Tara M. Madhyastha, Malayka J. Mottarella, and Chu Hsuan Kuo. 2020. Relating natural language aptitude to individual differences in learning programming languages. *Nat. Sci. Rep.* 10, 3817 (2020).

[112] Ruixiang Qi and Davide Fossati. 2020. Unlimited trace tutor: Learning code tracing with automatically generated programs. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. ACM, New York, NY, 427–433.

[113] Yizhou Qian, Susanne Hambrusch, Aman Yadav, Sarah Gretter, and Yue Li. 2020. Teachers' perceptions of student misconceptions in introductory programming. *J. Educ. Comput. Res.* 58, 2 (2020), 364–397.

[114] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.* 18, 1 (2017), 1–24.

[115] Kyle Reestman and Brian Dorn. 2019. Native language's effect on Java compiler errors. In *Proceedings of the 15th International Computing Education ResearchWorkshop (ICER'19)*. ACM, New York, NY, 249–257.

[116] Alexander Repenning. 2017. Moving beyond syntax: Lessons from 20 years of blocks programming in AgentSheets. *J. Vis. Lang. Sentient Syst.* 3 (2017), 68–91.

[117] Anthony Robins. 2010. Learning edge momentum: A new account of outcomes in CS1. *Comput. Sci. Educ.* 20, 1 (2010), 37–71.

[118] Anthony V. Robins. 2019. Novice programmers and introductory programming. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins (Eds.). Cambridge University Press.

[119] Philip M. Sadler, Gerhard Sonnert, Harold P. Coyle, Nancy Cook-Smith, and Jaimie L. Miller. 2013. The influence of teachers' knowledge on student learning in middle school physical science classrooms. *Am. Educ. Res. J.* 50, 5 (2013), 1020–1049.

[120] Jorma Sajaniemi and Marja Kuittinen. 2008. From procedures to objects: A research agenda for the psychology of object-oriented programming in education. *Hum. Technol.* 4, 1 (2008), 75–91.
[121] Jean Salac and Diana Franklin. 2020. If they build it, will they understand it? Exploring the relationship between student code and performance. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'20)*. ACM, New York, NY, 473–479.
[122] Igor Moreno Santos, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Experiences in bridging from functional to object-oriented programming. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. ACM, 36–40.
[123] Scala-lang.org. 2018. Making Scala Easier and More Popular, with Compiler Level Switches. Retrieved July 29, 2020 from from https://users.scala-lang.org/t/making-scala-easier-and-more-popular-with-compiler-level-switches/3534/12.
[124] Carsten Schulte. 2008. Block model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the 4th International Workshop on Computing Education Research (ICER'08)*. ACM, 149–160.
[125] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here we go again: Why is it difficult for developers to learn another programming language? In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. ACM, 1–11.
[126] Lee S. Shulman. 1986. Those who understand: Knowledge growth in teaching. *Educ. Res.* 15, 2 (1986), 4–14.
[127] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM Press, New York, NY, 378–389.
[128] Teemu Sirkiä. 2014. Exploring expression-level program visualization in CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling'14)*. ACM, 153–157.
[129] Teemu Sirkiä. 2016. Jsvee & Kelmu: Creating and tailoring program animations for computing education. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT'16)*. IEEE, 36–45.
[130] David Canfield Smith, Allen Cypher, and Larry Tesler. 2001. Novice programming comes of age. In *Your Wish Is My Command*. Elsevier, 7–19.
[131] Juha Sorva. 2008. The same but different—Students' understandings of primitive and object variables. In *Proceedings of the 8th Koli Calling International Conference on Computing Education Research (Koli Calling'08)*. 5–15.
[132] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Trans. Comput. Educ.* 13, 2 (2013), 1–31.
[133] Juha Sorva. 2018. Misconceptions and the beginner programmer. In *Computer Science Education: Perspectives on Teaching and Learning in School*, Sue Sentance, Erik Barendsen, and Carsten Schulte (Eds.). Bloomsbury, 171–187.
[134] Juha Sorva. 2020. Naïve conceptions of novice programmers. In *Computer Science in K-12: An A-To-Z Handbook on Teaching Programming*, Shuchi Grover (Ed.). Edfinity, 143–157.
[135] Juha Sorva. 2020. Programming 1. Retrieved June 2, 2021 from https://plus.cs.aalto.fi/o1/.
[136] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.* 13, 4 (2013), 1–64.
[137] Juha Sorva and Otto Seppälä. 2014. Research-based design of the first weeks of CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling'14)*. ACM, 71–80.
[138] Lisan Sulistiani and Oscar Karnalim. 2018. An embedding technique for language-independent lecturer-oriented program visualization tool. *EMITTER Int. J. Eng. Technol.* 6, 1 (2018), 92–104.
[139] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming misconceptions for school students. In *Proceedings of the 14th International Computing Education Research Workshop (ICER'18)*. ACM, New York, NY, 151–159.
[140] Ivan Tomek, Tomasz Muldner, and Saleem Khan. 1985. PMS-A program to make learning Pascal easier. *Comput. Educ.* 9, 4 (1985), 205–211.
[141] David S. Touretzky, Christina Gardner-McCune, and Ashish Aggarwal. 2016. Teaching "Lawfulness" with kodu. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM Press, New York, New York, 621–626.
[142] David S. Touretzky, Christina Gardner-McCune, and Ashish Aggarwal. 2017. Semantic reasoning in young programmers. In *Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE'17)*. ACM, New York, NY, 585–590.
[143] Ethel Tshukudu and Quintin Cutts. 2020. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the ACM Conference on International Computing Education Research (ICER'20)*. ACM, 227–237.
[144] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the tracing of recursion in the substitution notional machine. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*. ACM.

[145] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. *SIGCSE Bull.* 39, 3 (2007).
[146] Stella Vosniadou (Ed.). 2008. *International Handbook of Research on Conceptual Change* (1st ed.). Routledge.
[147] Jane Waite, Karl Maton, Paul Curzon, and Lucinda Tuttiett. 2019. Unplugged computing and semantic waves: Analysing crazy characters. In *Proceedings of the 1st UK & Ireland Computing Education Research Conference (UKICER'19).* ACM, New York, NY.
[148] David Weintrop, Alexandria K. Hansen, Danielle B. Harlow, and Diana Franklin. 2018. Starting from scratch: Outcomes of early computer science learning experiences and implications for what comes next. In *Proceedings of the 14th International Computing Education ResearchWorkshop (ICER'18).* ACM, New York, NY, 142–150.
[149] Greg Wilson. 2018. *How to Teach Programming (and Other Things)* (3rd ed.). Lulu Press.
[150] Greg Wilson. 2018. Is This a Notional Machine for Python? Retrieved February 3, 2021 from https://third-bit.com/2018/04/12/notional-machine-for-python/.
[151] Aman Yadav and Marc Berges. 2019. Computer science pedagogical content knowledge: Characterizing teacher performance. *ACM Trans. Comput. Educ.* 19, 3 (2019), 1–24.

Received July 2020; revised February 2021; accepted June 2021