# Programming Paradigms and Beyond

Shriram Krishnamurthi and Kathi Fisler
Brown University
sk@cs.brown.edu and kfisler@cs.brown.edu

Draft of a chapter from
*The Cambridge Handbook of Computing Education Research*, 2019
Sally Fincher and Anthony Robins, eds.

## Abstract

Programming is a central concern of computer science, so its medium—programming languages—should be a focus of computing education. Unfortunately, much of the community lacks useful tools to understand and organize languages, since the standard literature is mired in the ill-defined and even confusing concept of paradigms.

This chapter suggests the use of notional machines, i.e., human-accessible operational semantics, as a central organizing concept for understanding languages. It introduces or re-examines several concepts in programming and languages, especially state, whose complexity is understood well in the programming languages literature but is routinely overlooked in computing education. It identifies and provides context for numerous open problems worthy of research focus, some of which are new twists on long-running debates while others have not received the attention in the literature that they deserve.

# Table of Contents

## Introduction and Scope

Programming is central to computing. It is both the practical tool that actually puts the power of computing to work, and a source of intellectual stimulation and beauty. Therefore, programming education must be central to computing education. In the process, instructors need to make concrete choices about which languages and tools to use, and these will depend on their goals. Some emphasize the view of programming as a vocational skill that students must have to participate in the modern digital economy. Some highlight programming as an exciting creative medium, comparable to classical media such as natural language, paint, and stone. Irrespective of the motivation for teaching programming, from a professional computing perspective, programs are a common medium for representing and communicating computational processes and algorithms.

Different languages have different affordances. As many people have remarked, programming languages are a human–computer interface. At the same time, programming languages are executed by computers, which work with an unyielding logic that is very different from that of most human-to-human communication. This tension between human comprehension and computer interpretation has manifested in many CEdR studies over the years. Indeed, navigating that tension (and helping students learn to do the same) is one of the great challenges of computing education, and hence CEdR.

Computing educators need a lexicon and criterion through which to make, discuss, and teach choices about programming languages. Historically, much of our vocabulary has centered around a notion of "paradigms" that clusters languages by a combination of programming style and language behavior.[1] However, as programming languages and our technical understanding of them evolve, this notion is harder to maintain. This chapter therefore moves beyond paradigms to more nuanced ways of discussing languages, both amongst educators and with our students. We examine criteria and models for understanding languages through the lens of prior computing education research, with an eye towards exciting **open problems** (which will be flagged in boldface) that can take the field forward.

This chapter is written for aspiring researchers in computing education. It pushes for a deeper understanding of languages than at at the level of syntax (though syntactic mistakes, like using = for assignment, deserve their own opprobrium), pointing out that a given syntactic program can behave in several different ways (sec. 1). Understanding those behaviors—often captured in the concept of a notional machine (sec. 4)—is essential to making sense of the power and affordances of languages (sec. 5). The chapter discusses several long-running debates in the design and use of languages (sec. 6), including several other topics that remain wide open for researchers to investigate (sec. 7) that impact education. Rather than a single section of open questions, these are peppered throughout the chapter in context.

[1] In this chapter we use the term *behavior* to loosely refer to how a program executes, and *semantics* only to refer to a precise, formal description of behavior.

# 1 Beyond Syntax: Discussing Behavior

A lot of discussion about programming languages inevitably centers around syntax, because this is the most easily visible aspect of the language and the one that programmers primarily manipulate. A major challenge in programming, however, is mapping syntax to behavior, because computer behavior is hard to see and control. Furthermore, the ultimate goal is to create programs that behave in a particular way. Therefore, we will discuss many behavioral aspects of programs in this chapter.

We can make our discussion more concrete if we can refer to example programs. This section distills these into two examples. Consider the following two programs written in a generic syntax. For each one, predict what answer will be printed at the end of program execution:

Program 1:

```
y = 0
x = y + 5
y := 7  // change y
print(x)
```

Program 2:

```
o = {x: 1, y: 12}  // o is an "object" with two fields, x and y

procedure p(v):
  v.x := 5
end

p(o)
print(o.x)
```

Now that you have made your prediction, please revisit each program and ask whether it could reasonably produce a *different* value: what would that value be and how might it come about?

Stop! Did you make your predictions? If not, go back and do so first!

Most readers would expect the first program to print `5` and the second to also print `5`, because this is the outcome produced by the corresponding programs in languages like Java and Python. However, here are alternate explanations:

1. In the first program, `x` is defined to be `5` more than the value of `y` *whatever that value might be*. Thus, when `y` changes to be `7`, the value of `x` automatically updates to `12`.
2. In the second program, the modification to the `x` field is strictly local to `p`. When computation returns from `p`, the value of `o` is left unchanged (most probably, `p` received and changed a *copy* of `o`). Thus, the program prints `1`.

These alternate behaviors might strike some readers as natural but others as eccentric, so let us discuss them in more depth.

In Program 1, most readers probably find the answer `5` so obvious as to not even consider another possible outcome. Yet, when presented with the same program in a *spreadsheet*—where the variable become cells, `=` is a reference, and assignment to `y` becomes an update to its cell—nobody would expect anything but the *other* behavior, where `x` becomes `12`. Yet the spreadsheet behavior of automatically updating references is also found in textual programming languages, such as reactive languages (Bainomugisha et al., 2013). (It's worth noting that some of the odd student expectations mentioned in Pea's "Superbug" paper (1986) can just as well be interpreted as students expecting the language to behave reactively.)

In Program 2, the behavior comes down to a question of *aliasing* (Smaragdakis and Balatsouras (2015) provide a useful summary): when is a name (here, `v`) just an alias for another (here, `o`). Numerous studies have found that students (and even professional programmers) do not have a clear understanding of aliasing (Fleury, 1991; Ma, 2007; Tunnel Wilson, Fisler & Krishnamurthi, 2017) and that languages behave in ways that confound their expectations (Miller et al., 2017; Tunnel Wilson, Pombrio & Krishnamurthi, 2017). Yet aliasing is a crucial issue in programming, and one we will return to later in this chapter (sec. 5.2).

What these examples show is that *syntax does not inherently determine behavior*, as others have noted in the past (Plum, 1977; Fitter, 1979). Furthermore, though experienced programmers may have been primed to expect certain behaviors, (a) novices are not necessarily so primed (the reactive version of the first program shows that), and (b) even experienced programmers may not all agree (Tunnel Wilson, Pombrio & Krishnamurthi, 2017)—the aliasing disagreement in the second program shows that. Irreconcilable variation between syntax and behavior is par for the course, independent of efforts to make them align better in particular cases (Stefik & Siebert, 2013). Therefore, we need to investigate language *behavior* as a topic beyond the bounds of syntax.

## 2 Paradigms as a Classical Notion of Classification

To structure the study of languages, many authors have used the notion of "paradigm". Paradigms are supposedly groups that differentiate one class of similar languages from others in some high-level way, usually focused on features that exhibit common behaviors. Authors conventionally list a few major paradigms: imperative, object-oriented (henceforth "OO"), functional (henceforth "FP"), and logic, and other authors tend to add one or more of scripting, Web, database, and/or reactive.

We have already seen an example of reactive programming in the alternate interpretation of Program 1: in reactive languages, the program expresses *dependencies* but the language handles updating the values of variables in the presence of mutation. Similarly, in logic and database paradigms, programs express logical dependencies between elements of data, but determining answers is done through an algorithm that is hard-coded into the language. (Some

authors dub all these as “declarative” languages and imply that programmers can think only about the logic of problem-solving independent of algorithms, but in reality programs often have to be modified to adjust to the algorithms built into the language—sometimes even to ensure something as basic as termination.)

OO is a widely-used term chock-full of ambiguity. At its foundation, OO depends on *objects*, which are values that combine data and procedures. The data are usually hidden (“encapsulated”) from the outside world and accessible only to those procedures. These procedures have one special argument, whose hidden data they can access, and are hence called *methods*, which are invoked through *dynamic dispatch*. This much seems to be common to all OO languages, but beyond this they differ widely:

- Most OO languages have one distinguished object that methods depend on, but some instead have *multimethods*, which can dispatch on many objects at a time.
- Some OO languages have a notion of a *class*, which is a template for making objects. In these languages, it is vital for programmers to understand the class-object distinction, and many students struggle with it (Eckerdal & Thune, 2005). However, many languages considered OO do not have classes. The presence or absence of classes leads to very different programming patterns.
- Most OO languages have a notion of *inheritance*, wherein an object can refer to some other entity to provide default behavior. However, there are huge variations in inheritance: is the other entity a class or another (*prototypical*) object? Can it refer to only one entity (*single-inheritance*) or to many (*multiple-inheritance*), and if the latter, how are ambiguities resolved? Is what it refers to fixed or can it change as the program runs?
- Some OO languages have types, and the role of types in determining program behavior can be subtle and can vary quite a bit across languages.
- Even though many OO aficionados take it as a given that objects should be built atop imperative state, it is not clear that one of the creators of OO, Alan Kay, intended that: “the small scale [motivation for OOP] was to find a more flexible version of assignment, and then to try to eliminate it altogether”; “[g]enerally, we don’t want the programmer to be messing around with state” (Kay, 1993).

In general, all these variations in behavior tend to get grouped together as OO, even though they lead to significantly different language designs and corresponding behaviors, and are not even exclusive to it (e.g., functional closures also encapsulate data). Thus, a phrase like “objects-first” (sec. 6.1) can in principle mean dozens of wildly different curricular structures, though in practice it seems to refers to curricula built around objects as found in Java.

Finally, FP also shows up often in education literature, popularized by the seminal book by Abelson and Sussman (1985). In FP, programmers make little to no use of imperative updates. Instead, programs consume and produce *values*, and programming is viewed as the arrangement of functions to compose and decompose values (some have even dubbed FP as “value-oriented programming”). Due to the lack of mutation (sec. 4.1), aliasing problems (sec. 1) are essentially non-existent. FP is characterized by two more traits: the ability to pass functions as values, which creates much higher-level operations than traditional loops (an issue that

manifests in plan composition (sec. 6.3)), and tail-calls, which make loop-like recursive solutions just as efficient as loops, thus enabling recursion as a primary, and generalizable, form of looping. The two main variations in FP are whether the language is typed or not, and whether computation is eager or lazy, each of which leads to a significant difference in language flavor and programming style.

# 3 Beyond Paradigms

While paradigms have been in use for a long time, it is worth asking what they contribute to our understanding. First of all, should we view paradigms as classifiers that lump languages into exclusive bins? They are certainly interpreted that way by many readers, but two things should give us pause:

- As we saw in Program 1, being imperative does not preclude being reactive. Therefore, these cannot be viewed as independent. Reactivity simply gives an additional interpretation to an imperative update; furthermore, functional reactive languages have reactive behavior without explicitly imperative features, and reactivity can also be added to objects (Bainomugisha et al. (2013) provide a useful survey). Put differently, while OO and FP are statements about program *organization*, reactivity is a statement about program *behavior* on update. These are essentially orthogonal issues, enabling reactivity to be added to existing languages.
- Languages do not organize into hierarchical taxonomies the way plants and animals do; they are artificial entities that can freely be bred across supposed boundaries. Language authors can pick from several different bins when creating languages, and indeed modern mainstream languages are usually a mélange of many of these bins. Even archetypal OO languages like Java now have functional programming features (Gosling et al., 2015).

Furthermore, the paradigm bins sometimes conflate syntax and behavior. For instance, some authors now think of visual, block-based languages as a paradigm; however, blocks are purely a matter of program syntax and construction, whereas the other paradigms are about behavior. Indeed, block interfaces exist for imperative, OO, and FP. Thus it is unclear whether blocks should even be listed as a paradigm—which further highlights the confusion that this term creates.

Another source of confusion is whether “paradigms” are statements about programming *languages* or about programming *styles*. For instance, one might argue that they are programming in an “FP style” in a language that is not normally thought of as “functional”. These claims have a little validity, but must be viewed with some skepticism. For instance, a C programmer who is passing around function pointers is simulating a superficial level of functional programming, but in the absence of automatic closure construction and corresponding garbage collection, this is a weak and often unsatisfying simulation. Similarly, Python’s lack of tail call elimination makes many natural FP patterns unusable in practice. Nevertheless, the possibility of such simulations makes it even harder to understand what paradigms are.

Yet another source of confusion is between the language and the operating environment. In a "batch" program, the program has a well-defined beginning and end. It may periodically pause to accept input, but the program determines when that happens. In contrast, in an event-driven program, it is the operating environment that is in charge; each event (whether a keystroke, a screen touch, a Web request, a network packet arrival, or the tick of a clock) causes some part of the program to run in response. After responding to the event, the program (usually) returns to quiescence, and "wakes up" on the next event. Such a program has no well-defined beginning or end, and in principle runs forever. The programmer's challenge is to arrange how state is transferred between the events. This can be done in several ways: imperatively, using objects, functionally, reactively, and so on. Thus event-driven programming is another cross-cutting notion that is independent of and orthogonal to the *program*'s organization—rather, it is a statement about the program's relationship with its *operating environment*. Lacking a clear definition of "paradigm", it is therefore entirely unclear whether even-driven-ness is one.

Thus, even though paradigms are widely used by authors and in the literature, some authors—especially in the programming languages research community—question or even reject their use (Krishnamurthi, 2008). Our examples and discussion points illustrate why a focus on *behavioral* properties and features provides a more meaningful framing.

## 4 Notional Machines

Whereas a programming-languages researcher captures program behavior through semantics, computing education researchers instead use the idea of *notional machines* (du Boulay, 1986; Sorva, 2013; Guzdial, 2015). A notional machine is a crisp, human-friendly abstraction that explains how programs execute in a given language or family of closely-related languages—i.e., a model of computation. While notional machines are usually viewed as a tool for learning to write and trace programs, they are also a useful way for us to think about language classification: essentially, 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. Seen this way, many eager FP languages have very similar notional machines (and may even share implementations); many "scripting" languages have strong similarities in some respects (but notable differences in others); while many OO languages actually have significantly different notional machines (for instance, a semantics of Java (Flatt, Krishnamurthi & Felleisen, 1998) and of JavaScript (Guha, Saftoiu & Krishnamurthi, 2010) are vastly different). There can be many notional machines for a given language, reflecting different goals, degrees of sophistication, levels of abstraction, and so forth.

Novice programmers do not infer accurate notional machines just from writing programs, as established from the literature on misconceptions. They sometimes see a program just as syntactic instructions, without a clear sense of how the instructions actually control some underlying device (du Boulay, 1986). The idea that a program controls a device, and thus has dynamic behavior, has been identified as an essential concept in learning about programming (Schwill, 1994; Shinners-Kennedy, 2008). Notional machines concretize this idea in a specific behavioral model. This suggests that teaching about notional machines and semantics is

important (though some who take an extreme constructivist perspective might disagree (Greening, 1999)). Sorva's (2013) survey article on notional machines covers various theoretical framings for why students need to learn a semantics.

Despite some noticeable treatment in the literature, notional machines do not feature prominently in curricula or texts for computing courses. While execution models are often implicit in visualization and debugging tools (Hundhausen, Douglas & Stasko, 2002; Naps et al., 2003; Sorva, 2012; Kölling, Brown & Altadmri, 2015), few papers present notional machines explicitly alongside teaching programming constructs. Research suggests, however, that teaching notional machines early could have significant value. Cognitively, building a new mental model (in this case, of program execution) is significantly easier than updating an existing (flawed) one (Schumacher & Czerwinski, 1992; Slotta & Chi, 2006; Gupta, Hammer & Redish, 2010). Building accurate and effective models is more likely to occur through activities that engage explicitly with the content (in contrast to passively working with visualizations, for example) (Kessel & Wickens, 1992; Savery & Duffy, 2001; Freeman et al., 2014). Practically, Nelson et al. taught students tracing through a notional machine at the very start of a programming course (Nelson, Xie & Ko, 2017). These students performed better on the SCS1 test of CS knowledge (Tew, 2010) than students who did code-writing tutorials rather than tracing. diSessa and Abelson (1986) leveraged UI design to convey aspects of the notional machine (such as scoping) with their Boxer system, noting that learning a notional machine is a necessary challenge. They also raised questions about which notional machine to teach to end-users who are learning programming in order to control computational media at a small scale, compared to those preparing for professional software practice.

Research evidence on which notional machines to teach and when is sorely lacking in the computing education literature. Some work has evaluated specific models while checking for specific student misunderstandings (Ma, 2007; Tessler, Beth & Lin, 2013; McCauley et al., 2015), but this body of research is not substantial enough to support general conclusions. Furthermore, there is scant attention to how notional machines evolve as students gain sophistication in upper-level computing courses. Sometimes, upper-level students even need to correct inaccurate models that formed in earlier courses (Tunnell Wilson, Krishnamurthi & Fisler, 2018); mechanisms for doing so are an **open problem** that need attention in computing education research.

## 4.1 The Challenge of Mutation and State

Notional machines are a useful lens through which to explore the complexities of reasoning about state (program behavior in the presence of mutation). Stateful programming has been taken by many as a sine qua non of programming education (for instance, it is virtually never mentioned as an assumption or threat to validity or generalizability). At the same time, numerous studies show that students struggle with this concept (du Boulay, 1986; Goldman et al., 2010; Sirkiä & Sorva, 2012), both as novices and as upper-level students (through interactions between state and other language features) (Tunnell Wilson, Krishnamurthi & Fisler, 2018), while students working in non-stateful paradigms sometimes perform well on

problems that are challenging in stateful contexts (Fisler, Krishnamurthi & Siegmund, 2016). Taken together, these observations make comparative studies between stateful and non-stateful features one of the most significant **understudied** topics in computing education.

If state is so challenging for students to learn, why is it so popular in introductory contexts? Purely as a language feature (setting aside pedagogy for a moment), state has many benefits:

- It provides cheap communication channels between different parts of a program.
- It trades off persistence for efficiency.
- It appears to have a relatively straightforward notional machine, which lends itself to familiar-looking metaphors (like “boxes”).
- It corresponds well to traditional control operations like looping.

However, these benefits of state become far less clear once we broaden our scope beyond very rudimentary programming. For instance:

- State introduces time and ordering (as Program 1 reveals) and forces students to think about them.
- State reveals aliasing (as Program 2 reveals). Aliasing is particularly problematic in the case of parallelism/concurrency, which is increasingly a central feature in programming.

State thus requires a complex notional machine to account for all these factors. The apparent simplicity of stateful notional machines arises because most computing education literature usually just ignores some of these features (e.g., compound data with references to other compound data). This results in notional machines that are not faithful to program behavior, and hence are either useless or even misleading to students.

Thus, we believe it is worthwhile to revisit our basic assumptions about stateful programming’s role in education: perhaps as an *advanced* introductory concept rather than as the most introductory one. State is a powerful tool that must be introduced with responsibilities. In short, while the community continues to have a debate about “objects-first” (sec. 6.1), it is also worth having a “state-first” debate. Making progress on this issue is an **open challenge** in this field, but it will require educators with deep-seated beliefs to be willing to reexamine them.

It is worth noting that such shifts in conventional wisdom are feasible. For decades, automatic memory management (often called garbage collection) was considered a fringe feature, and most mainstream languages did not offer it. Students were therefore forced to confront memory management operations relatively early in their curriculum. As garbage collection has become widespread, and the problems caused by poor manual memory management have become better known, this topic has moved to more advanced courses, where it is hopefully taught with more care. The growing understanding, in industrial practice, of the problems with unfettered state may therefore similarly result in many more “state-later” curricula.

## 4.2 Notional Machines for Related Disciplines and Transfer

When computing is used or taught in conjunction with another discipline (a model frequently proposed to scale pre-college computing education (Stanton, et al., 2017)), it is important for the computing curricula to align well with the (sometimes implicit) behavioral models used in the

other discipline. This seems necessary both to avoid confusing students and to eventually achieve transfer. For instance, using programming to teach algebra suggests programming with a notional machine that matches what students see in their math classes: one whose functions pass the "vertical line test" (i.e., they truly are functions), whose function application corresponds to algebraic substitution, whose variables behave like algebraic variables rather than stateful ones, and so on. Similarly, in physics, it is standard to model systems in terms of differential equations, which suggests different models of reactive programming (Felleisen et al., 2009) than traditional imperative event-driven code. In a data science curriculum, it may be important to first acquaint students with query operations before loops. It is therefore not only important to adjust the notional machine of computing to match those of other disciplines in interdisciplinary contexts, it is also worth wondering whether insights about computational models from other disciplines may lead to a less onerous study path for students of computing.

# 5 Human-Factors Issues

Thus far, our discussion of languages has taken a technical perspective, framed largely by semantic behavior. Some debates about paradigms have invoked human-factors issues, such as whether one paradigm provides a more "natural" way to think about programs (Lister et al. 1986; Wiedenbeck, et al., 1999) or whether some constructs align better than others with how novices conceive of computations (Miller, 1981). Human-factors questions arise around each of syntax, behavior, and integrated development environments (IDEs) for programming languages, both in isolation and in their interactions (Kelleher & Pausch, 2005). Programming involves many tasks—reading, writing, debugging, and modifying code—and each has its own human-factors nuances. Furthermore, not all programming education has the same end-goal: different environments are designed to, for instance, help novices with mainstream languages (e.g., Greenfoot, https://www.greenfoot.org/), prepare students for professional practice (e.g., Eclipse, http://www.eclipse.org/), motivate students to play with computing (e.g., Scratch (Resnick et al., 2009)), or help students visualize program behavior dynamically as code is edited (e.g., Learnable Programming (Victor, 2012)). A meaningful discussion of programming languages in educational contexts must account for human-factors issues, including the differing goals and abilities of the target student audience.

## 5.1 Interference between Human Language and Behavior

Programming languages use terminology from human languages (such as "if" or "while") that suggest the desired behavior of particular constructs. In practice, programming languages have more precise semantics for these terms than does human language. Many early and recent studies have identified gaps between programming semantics and humans' intuited semantics. For example:

- Pea's classic "Superbug" paper (1986) proposed that students ascribe an intelligent "hidden mind" to program execution that would make programs behave in accordance with human communication patterns.
- Soloway, Bonar and Erlich (1983) showed differences in how novices interpreted program results based on different structural organizations of `repeat` loops in Pascal.

More recent studies by Stefik and his colleagues show confusion due to keyword choices (Stefik & Gellenbeck, 2011) and leverage user-studies to design constructs to be more intuitive for novices (Stefik & Siebert, 2013). Most of these works identified problems with the specification of control-flow in programs.

- Miller's studies of how non-programmers expressed computations (Miller, 1974; Miller, 1975) highlighted various ways in which complex control-flow raised significant differences between human languages and programming languages: among the issues he cited were the need to specify initialization and exceptional cases in programming (neither of which are common in everyday human communication), and the precision required to bound iteration in programming (where human language often leaves such bounds vague).

Pane's dissertation (2002) provides more extensive coverage of this topic, including survey instruments and details results from multiple user studies (some of which worked with children).

## 5.2 Inferring Mental Models of Notional Machines

Interference between natural language and programming semantics matters because it affects users' construction of mental models of program execution. Ideally, programming education would help users form an accurate model of a notional machine for the language they are using. In practice, programmers often construct mental models of computation that are inconsistent with the actual behavior. This inconsistency seems unavoidable for many reasons. For instance, the mapping from the surface syntax to intended behavior may not be obvious or may have multiple reasonable interpretations (see the programs in sec. 1). In addition, programmers develop their understanding of languages incrementally (Findler, et al., 2002, sec 3.1), and their initial models may clash with some later behavior they have not yet learnt. Sorva's detailed review of literature on notional machines and mental models explains that "a novice's mental model of a notional machine is likely to be—typically of mental models in general—incomplete, unscientific, deficient, lacking in firm boundaries, and liable to change at any time" (Sorva, 2013, p. 8-10).

Inaccuracies in inferred mental models reflect *misconceptions* about program execution. The computing education literature abounds with evidence and descriptions of misconceptions about program execution, particularly among novice programmers. An entire chapter of this handbook focuses on this topic. From the perspective of this chapter, misconceptions are interesting because they can manifest differently across programming languages or notional machines for those languages.

Understanding how different notional machines for the same language impact formation of mental models of program execution is a significant **open problem**. The potential implications are even more interesting: if some notional machines prove easier or more robust for students to learn at first, should that affect the order in which we introduce language features, even if it means moving away from the models that currently underlie most introductory curricula?

Different notional machines may also demand different cognitive loads; we are **not aware of any existing research** that contrasts the cognitive loads or demands of different notional machines for different programming-related activities (such as authoring versus debugging). In fact, we **conjecture** that the cognitive loads of different styles of programming may vary by activity: it may be that imperative programming is much easier for writing programs quickly (since state provides convenient communication channels between parts of a program), but these same benefits make subsequent reasoning and debugging much harder; functional programming may have the opposite affordances.

These questions of cognitive load are not just a question for novice programmers: aliasing, for example, has proven difficult to reason about even for professional programmers working with parallel programming (Cooper et al., 1993). Understanding these issues along all these different dimensions should give us a much more nuanced and sophisticated understanding of the impact of paradigms, programming styles, and notional machines than we currently find in the literature (where, for instance, it is very uncommon to find a paper that asks students to explain their own programs six months later and see how well they are able to fare).

Research is also needed well beyond the early curriculum. Student misconceptions about programming concepts can persist into the upper-level curriculum: one recent study found juniors and seniors holding substantial misconceptions about aliasing in Java, despite several programming courses (and other experiences) that used the language (Fisler, Krishnamurthi & Tunnell Wilson, 2017). As students move beyond the first year or into jobs, they often learn new programming languages. Do students take their old notional machines with them, even if the new languages have different features that contradict the behavior of the notional machine that they knew? How do we teach students to adapt existing models to accommodate new features? How do we migrate students to an entirely new notional machine when needed (such as when moving from non-reactive to reactive programming), or when starting to confront parallelism? There are significant **open problems** and opportunities for research that explores how notional machines and students' informal programming models can and do evolve across languages, features, and complexity of problems.

## 5.3 Visual and Blocks-based Languages

Blocks-based or visual notations provide a significantly different user interface than textual languages. These notations do not inherently correspond to new language features. Visual notations can, however, have rich interactions with behavior that affect how users interact with or evaluate programs. Early work in visual languages used spatial relationships among syntactic features to convey semantic relationships between concepts and constructs (Haarslev, 1995). Different visual notations were suited to reflecting different semantic relationships. In some languages (Kahn & Saraswat, 1990), program execution was reflected in transformations of the visual notation (similar in spirit to a what substitution model (Plotkin, 1975; Felleisen & Hieb, 1992) allows); in such cases, the program syntax also provided syntax for the state of the notional machine. Sorva's dissertation (2012) provides a detailed discussion of tools and

taxonomies for integrating visual representations with behavior or program evaluation in a pedagogical context.

Blocks languages such as Scratch, Alice, Snap!, and Blockly use visual representations to simplify syntax and convey grammatical features, rather than to illustrate behavior or program execution. Blocks syntax for imperative languages, for example, uses different shapes or connectors to distinguish statements (such as assignment operators or control flow) from value-producing expressions. In contrast, blocks syntax for a functional language might use color or shape to distinguish datatypes, restricting block nesting to cases in which the expected and provided types match up (Code.org, n.d.; Vasek, 2012; Schanzer, Krishnamurthi & Fisler, 2015). Blocks typically introduce an additional syntactic layer between the programmer and an underlying textual language (used for compilation or interpretation): this provides expressive flexibility for designers of blocks languages to create constructs or keywords from natural-language phrases that may not exist in an underlying textual language (Weintrop & Wilensky, 2015a).

Multiple studies contrast blocks and textual languages and find differences with respect to misconceptions, program comprehension, and learning outcomes. Weintrop and Wilensky (2015b; 2017a) tested high-school students on program-tracing questions given in each of Java syntax and Snap!. Students generally performed better with blocks. The reason, however, may be due to issues beyond the mere use of blocks. For example, Snap! labels its repetition block as `repeat`, while Java uses `for`: studies of loop-naming in textual languages suggest that `for` is not an intuitive term for repetition among novice programmers (du Boulay, 1986; Stefik & Gellenbeck, 2011). A followup study contrasted student work using isomorphic blocks and textual environments; students in the blocks condition performed better on a content assessment than those in the text condition (Weintrop & Wilensky, 2017a). This study identified some concrete affordances of blocks. For example, students were more likely to (correctly) assert that only one branch of an `if`/`else` expression would be evaluated when working in block syntax. This may be due to the shapes of blocks, in which a multi-line statement or expression is part of a single visual element, rather than spread across separate lines of text. Other observations about misconceptions around variables and function calls appear *attributable* to block-syntax features (Weintrop & Wilensky, 2015b). Grover and Basu (2017) found misconceptions about variables and loops in their study of block syntax with middle-school students. Lewis (2010) contrasted attitudinal and learning outcomes between Scratch and Logo with 10-12 year-old novice programmers. The Scratch students showed improved performance at interpreting the behavior of conditionals, but not at interpreting loops. These studies reinforce that blocks do not inherently address long-standing challenges that students have with comprehending and evaluating core programming constructs.

Students who continue beyond an early blocks-based introduction to computing confront the transition to textual syntax. This transition is multifaceted: while the syntax clearly changes, the notional machine may change as well if the textual language has a richer or different feature set. In addition, the switch to textual syntax often accompanies an increased complexity in programming tasks. The program-construction habits that students developed for simple blocks

programs may not scale to larger problems, a problem that Meerbaum-Salant et al. (2011) observed with Scratch students. Powers, Ecott and Hirshfield (2007) tracked issues that arose as tertiary students transitioned from Alice to one of Java or C++. They hypothesized that differences in code organization between the Alice IDE (which uses different panels for different concepts) and textual code might explain the considerable difficulties that students demonstrated in writing and understanding the textual programs. Armoni, Meerbaum-Salant and Ben-Ari (2015) did not observe such problems when using Scratch as the initial blocks language with middle-school students (who later transitioned to C++ or Java). Their data show students performing better on questions about looping when working in Scratch (a result that contradics that of Lewis (2010)), though measures on other constructs were not statistically significant. These and other studies (Malan & Leitner, 2007; Grover & Basu, 2017) generally show blocks as having positive impact on some combination of performance, motivation, or retention (though not for all students, as authenticity can be a concern, as we discuss momentarily). The main take-away from these studies is that blocks have many affordances, but also some notable challenges in the larger scope of early computing education.

Several efforts attempt to ease the transition through hybrid environments that allow students to switch between blocks and textual notation: some hybrid tools interleave blocks and textual notation (Bau et al., 2015; Kölling, Brown & Altadrmi, 2015, Mönig, 2015), while others let students switch their view of the same code between blocks and textual syntax (Homer & Noble, 2014; Bau et al., 2015). Ongoing research into the affordances of each modality considers many criteria, including student learning gains, attitudes, and preferences (Weintrop & Wilensky, 2017b). Evidence suggests that the hybrid tools enable students to construct code more quickly and with fewer errors than in purely textual tools (Price et al., 2016), but each modality appears to support some tasks and goals better than the other (Weintrop & Wilensky, 2017b). In general, careful attention to pedagogy and design is needed to create tools through which students learn to migrate across notations (Dann et al., 2012).

Although much of the research on blocks focuses on students' initial programming experience, some work looks at the longer-term impacts of starting in blocks. Weintrop and Wilensky's (2017a) studies with high-school students showed that differences between starting in blocks versus text faded after 10 weeks of programming in Java. This suggests that blocks may not be as critical for all student audiences. Another concern arises from a human-factors perspective. Students gain more confidence, and sometimes interest, in programming when they perceive what they are learning as authentic (relative to their perceptions of programming practice) (Lewis et al., 2014). Both DiSalvo's (2012) Glitch Game Testers work and Weintrop and Wilensky's (2015a) studies with high-school students reveal that students sometimes view blocks as an inauthentic programming experience. Identifying an appropriate interplay of textual and visual notations in early programming education thus remains an important **open problem** for future research.

## 5.4 Accessibility of Program Authoring and Environments

Discussions about learning languages can easily mask hidden assumptions that all programmers are similarly abled. Programming notations, whether text or blocks, tend to rely on the ability to manipulate code visually. Drag-and-drop programming tools demand fine-motor skills. Cognitive loads from debugging could differ dramatically across users who process information differently, simply based on the nature or amount of information that an IDE expects a programmer to remember. A programmer may not know the human language or alphabet from which keywords are taken, thus depriving them of context. Supporting differently-abled users obviously depends on interface design for IDEs, but technical decisions regarding syntax and behavior can limit or enhance how interface designers support differently-abled users. In keeping with this chapter's focus on behavioral features, we focus this section on the interplay of technical decisions, interface decisions, and user abilities.

Most IDEs require users to leverage sight to navigate through and skim code, to scan a display of options in interfaces that support discoverability, and to validate code while typing. To communicate with visually-impaired users, IDEs use auditory information, often delivered through screen readers. Reading code one token at a time, however, provides too much information for high-level navigation and comprehension tasks (Stefik, 2008). Effective auditory tools instead attempt to convey high-level code structure (Baker, Milne & Ladner, 2015). Abstract syntax trees (ASTs) capture this structure, but ASTs are difficult to create for in-progress or buggy programs that don't properly parse (an issue that can arise with textual, but not blocks, syntax). So-called bicameral syntax (Krishnamurthi, 2006)—such as the parenthetical structure of Lisp-ish languages or the distinction between well-formedness and validity in XML—splits adherence to the grammar into two phases rather than one.This provides an intermediate level of lexical structure, which can enable navigation of code that satisfies the intermediate syntax even though it does not yet properly parse. Quorum (Stefik, 2008) uses particular keyword orders to direct screen readers to relevant tokens through which to convey code structure (without requiring bicameral syntax). Languages designed to interface with screen readers must consider the representation of program output as well as source code: if a program produces a tree, an image, or a video, for example, the screen reader must be able to explain all those outputs to a user (Bootstrap, 2017).

Research on instructional design (Mayer & Moreno, 2003) shows that users with full access to both their visual and auditory channels can process information from both channels simultaneously. Some tools (Stefik & Gellenbeck, 2009) provide both verbal and auditory information during program execution, finding that purely auditory tools are less effective for sighted users. Stefik, Hundhausen, and Patterson (2011) found that well-designed auditory cues could enable (sighted) users to approach the effectiveness of programmers working without visual cues. More novel approaches have attempted to create musical renditions to communicate program-execution behavior (Vickers & Alty, 2002). How to convey program execution and notional machines for differently-abled users is a significant **open question**.

Other substantial other work in IDE design impacts how differently-abled users construct (rather than read) programs. These include creating structured editors or simplified spoken-language commands to enable voice-driven code construction (Begel & Graham, 2005), internationalization support to customize menu entries, replacing English keywords with ones in other languages, or writing new languages for speakers of human languages other than English. These efforts are too broad to treat comprehensively in this chapter, but they represent the myriad ways in which designers of languages and tools can accommodate the needs of diverse users.

In short, language design for differently-abled users is an **open area**. Accessible interfaces cannot simply be bolted on to existing designs: language design determines what needs to be communicated in the first place, which in turn affects the mechanisms that can be used. Developing programs requires tasks beyond coding—such as tracing, debugging, and navigating documentation—that require tool support (Ko, Myers & Aung, 2004). Practitioners should consider the needs of users and students when selecting tools. Researchers should assess the abilities being assumed in their tool designs, both theoretically and with user studies. Such studies have value beyond supporting accessibility: tools that accommodate users with severe impairments often benefit significant numbers of users with milder, less visible, impairments of the same type, as well as non-impaired users (a user-interface design principle called Universal Design (Mace, Hardie & Place, 1991)).

# 6 Long-Running Debates and Questions

Many topics in computing education have been discussed for a long time and at great length, but remain unresolved. Several of these pertain to language features, either directly or indirectly. We briefly discuss a few of these that fit particularly well with the themes of this chapter.

## 6.1 Objects-First Debate

Introductory computer-science pedagogy largely used procedural languages (such as COBOL, Basic, Pascal, and C) until the late 1990s. As C++ and Java gained hold in industry, both industry partners and parents pressured schools to transition to teaching OO in CS1 (de Raadt, Watson & Toleman, 2002). Independently, some instructors began questioning whether OO programming could foster better software design skills than procedural programming in CS1 students (Decker & Hirshfield, 1994). The resulting educational interest in OO led to significant discussion of the affordances and limitations of *objects-first* or *objects-early* pedagogies and curricula. Although objects-early pedagogies frequently use Java (which really presents a *class*-oriented view of programming rather than a purely *object*-oriented one), many languages support programming with classes and objects. Different instructors have used the term “objects-early” for at least three different curricular goals: using objects, defining and implementing classes, and using other features of OO (such as inheritance) (Bennedsen & Schulte, 2007).

Proponents of objects-early claim that classes and objects effectively model real-life problems and interactions (Hu, 2004), which should enable introductory courses to motivate and situate content better than with procedural languages. Some researchers have questioned whether OO is indeed a natural way for humans to view systems (Pane, Ratanamahatan & Myers, 2001). Furthermore, not all data that get modeled through objects have real-world analogues. Skeptics of objects-first argued that focusing on objects early squeezes out time used for basic programming constructs (Reges, 2006) and algorithmic thinking and problem solving (Lister et al., 2006). Programming with objects can involve both basic imperative constructs as well as defining methods (functions), leading some to question whether OO demands more than either imperative or functional programming would (Cooper, Dann & Pausch, 2003; Sajaniemi & Kuittinen, 2008). A notional machine that supports objects is more complicated than one supporting only procedures over atomic data (Sajaniemi & Kuittinen, 2008). Different researchers have argued for different notional machines for objects: some based just on message passing, others on conventional models of state (Sorva, 2012). In 2004, a heated debate about objects-first erupted on the SIGCSE email list: Bruce (2005) summarized the discussion in an article that touches on teacher preparation, IDEs, order of content, OO learning goals, and other factors that faculty use in considering first programming languages. The summary also called for educational research to study questions about first programming languages more systematically.

Research on the objects-first debate largely attempted to compare objects to procedural programming for introductory students. Wiedenbeck and colleagues showed that students instructed in a procedural pedagogy were better at comprehending (short) programs than students instructed in an objects-early pedagogy, though students instructed in an objects-early pedagogy were better at understanding functions at the individual class level (Wiedenbeck & Ramalingam, 1999; Wiedenbeck et al., 1999). Yet other studies (Vilner, Zur & Gal-Ezer, 2007; Ehlert & Schulte, 2009) have shown that objects-first versus objects-later lead to similar learning gains. Students in these populations differ only in terms of perceived difficulty of topics. Additionally, the amount of scaffolding required for teaching objects early is greater than that for a procedural pedagogy (Nordström & Börstler, 2011). Some argue that instructors have to work harder in order to come up with pedagogically-appropriate examples when starting with objects (Nordström & Börstler, 2011). Ideally, studies would also examine the cognitive load of starting with objects. One study (with upper-level software engineering students) has shown that inheritance leads to increase mental overhead (Cartwright, 1998), but not all objects-first curricula address inheritance early on.

Objects are sometimes taught with visual environments that illustrate the behavior of programs. The Alice programming environment (www.alice.org) teaches programming in the context of microworlds, where each visible entity in the world corresponds to a object within an OO program. Cooper, Dann & Pausch (2003) argue that visualizations and simulations are an important technique for controlling the complexity that students would otherwise have to confront when working with objects early in their programming education.

Despite the amount of work published on both sides of the debate, there is no definitive

evidence that objects-early is better than procedural or some other approach (Bailie et al., 2003; Tew, McCracken & Guzdial, 2005; Sajaniemi & Kuittinen, 2008). One key challenge in studying this question arises because an objects-first curriculum may focus on different skills or styles of programs than curricula that start with other approaches: these differences can make it hard to run studies in which students from different approaches are asked to attempt the same questions. Given that studies have nevertheless compared objects-early to imperative counterparts, it is unclear why there are few if any studies against the long-standing tradition of teaching functional programming. Conducting them remains an **open problem**.

Similar questions about commensurability arose in earlier language-choice debates as well (Johnson, 1995; Brilliant & Wiseman, 1996). Social factors (such as which languages have currency in industry) may also affect linguistic choices in some settings. This is a different aspect of the debate which arises as people debate the rationale for broad efforts to expose all students to computing (Guzdial, 2015).

## 6.2 Repetition: Iteration, Recursion, and More

Repetition is a central concept in programming: processing a collection of data requires some form of repetition. A long-standing debate has been which forms of repetition to use. Most curricula focus heavily on *iteration*, as manifest in features like loops, while some curricula depend instead on *recursion*, characterized by procedures that directly or indirectly invoke themselves. A great deal of literature has been written about the differences between the two. Rather than recapitulate this literature, we instead highlight important issues that arise from a programming languages perspective, but have been largely overlooked in existing research.

- The study of recursion has been negatively impacted by inane examples such as Fibonacci and factorial functions, which have little computational value and confuse issues such as performance with the notion of recursion. In contrast, recursion arises naturally in the processing of recursive *data* such as trees or recursively-defined lists. Since many introductory curricula do not cover these data, the research literature on this topic is often missing the point of using recursion in the first place.
- Repetition often interacts in interesting ways with mutation. For instance, consider building a calculator application, which displays a panel of ten buttons, pressing each of which prints the corresponding digit. One might write the following loop:

  ```
  for (i = 0; i < 10; i ++):
    buttons[i].set-handler(lambda(): print i)
  ```

  Contrast this to the seemingly equivalent recursive solution:

  ```
  map(lambda(i): buttons[i].set-handler(print i),
      [0 .. 9])
  ```

  The latter programs works exactly as desired but the former program, when run in a

conventional imperative setting, is buggy: every button will result in the output `10`, because of aliasing (sec. 1). (The reader may find it instructive to consider why these programs behave as they do.)

- Even the study of recursion on recursive data has important distinctions. A *structurally* recursive solution follows the structure of the data precisely—just as a loop does over a linear structure—resulting in simpler code and predictable performance and termination. A *generatively* recursive solution computationally creates new sub-problems on recursive calls, and hence lacks the advantages of structural recursion (Felleisen et al., 2001). Therefore, structural recursion is much simpler and a gentler way to introduce students to recursion, while still enabling sophisticated programs. Unfortunately, many popular introductory recursive problems (such as fractals, Euclid's algorithm, Newton-Raphson, etc.) are generative, thereby creating artificial complexity that confuses the outcome of studies.

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*, which are inspired by the mathematical notation of set theory. Big-data programmers use abstractions such as MapReduce (Dean & Ghemawat, 2004) that consume high-level descriptions of behavior. The users of these abstractions do not perform explicit iteration; that is hidden inside the implementation of the abstraction, freeing it up to parallelize and otherwise optimize execution. Similarly, every database programmer using SQL makes extensive use of repetition patterns through queries. To an iterative thinker a query might look like loops and to a recursive thinker they might appear to be higher-order recursive operators, but the query writer does not write any explicit repetition at all. Thus, 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. (Curricula that use high-level operators before teaching iteration or recursion include Harvey and Wright's (1999) *Simply Scheme* and Bootstrap:Data Science (Bootstrap, 2018), both of which use high-level operators before teaching iteration or recursion.) As computing curricula increasingly embrace big data, the need for alternate and richer notions of repetition will grow in importance.

## 6.3 Plan composition

Soloway (1986) proposed the Rainfall problem in the early 1980's in the context of studying how students composed code for the subtasks of a problem into a cohesive program (a process he termed *plan composition*). Roughly, Rainfall asks for the average of non-negative numbers that occur in the input before some sentinel value (such as -999). Soloway's group introduced a rich vocabulary of *plans*, *goals*, *mechanisms*, and *explanation*s (Soloway, 1986). Their finding that students struggled to develop solutions to this problem spawned many subsequent efforts to understand students' successes and failures with so-called *plan composition*, with multiple studies corroborating the initial results (Ebrahimi, 1994; Simon, 2013).

By and large, the first 25 years of research on plan composition did not consider the influence of programming languages on how students structure code. Ebrahimi (1994) considered that language might make a difference, and had students trained in different programming languages solve Rainfall; the results were uniformly weak across languages. However, Ebrahimi's students worked within the imperative model in each language (including Lisp), rather than consider other models such as functional; this limited the potential impact of looking at different languages. In 2014, Fisler (2014) published the first study of Rainfall with students who were trained in functional programming (with higher-order functions and without assignment statements, using the *How to Design Programs* curriculum (Felleisen et al., 2001)). The students in her study performed much better than in prior studies; more interestingly, however, these students produced solutions with very different high-level structures and task clustering than in earlier studies. In particular, whereas students in earlier studies tried to solve the entire problem in a single traversal of the data (using a `for`- or `while`- loop), Fisler's participants often used built-in or higher-order functions to implement some subtasks. This created solutions that performed multiple passes over the input data, often to clean out unwanted data before computing the average.

Subsequent studies of Rainfall in diverse programming paradigms (Seppälä et al., 2015; Fisler, Krishnamurthi & Siegmund, 2016) suggest that languages affect planning in two ways: different languages have different idioms and provide different built-ins. FP teaches students to compose short functions for specific tasks; this style often creates intermediate data. With procedural programming, students often learn to create fixed-size arrays. For typical planning programs, the sizes of intermediate arrays are often only determined on the fly; this could steer students away from multi-traversal solutions. Students who know library functions that transform entire lists or strings may use them instead of creating loops to manually traverse these data structures as arrays; in a later study (Fisler, Krishnamurthi & Siegmund, 2016), students working in Java who knew such library functions produced multi-traversal solutions akin to those in functional programming. Iterators with task-specific or semantically-rich names, such as `map` and `filter`, may suggest different decompositions than generically-named constructs such as `for`. Efforts to teach novices named patterns (Muller, Ginat & Haberman, 2007) and strategies (de Raadt, Watson & Toleman, 2009) have shown promise in helping students implement multi-task programs from the plan-composition literature. The potential influence of built-in language constructs that provide these patterns has yet to be studied in detail. All of these points raise hypotheses about languages and planning that **warrant further investigation**. Such studies should also explore potential pitfalls to richer language support, such as the challenges of tracing higher-order control flows.

In summary, fine-grained details of languages and idioms—as well as problems that students have solved previously (Pirolli & Anderson, 1985; Spohrer & Soloway, 1989; Rist 1991)—all appear to impact how students plan programs. Researchers conducting or reporting studies in this area should include details on the code patterns students were taught, the built-in operators that they had seen, and where the planning problems fit into a course's larger curricular sequence. Merely reporting on the language used for a study does not provide sufficient context.

# 7 Some (Other) Open Questions

Some linguistic topics highly pertinent to computing education have not gotten the attention we believe they deserve. We introduce them below as a spur to future research.

## 7.1 Sublanguages and Language Levels

Consider a typical programming textbook. It begins with a small core of a language and gradually expands the size of the language as the student progresses in their learning. Yet most language implementations (compilers, programming environments, …) do not mirror this growth. As a result, a student may accidentally—even by means as innocent as a typo—stumble upon parts of the language they have not been taught. This can either result in a program "working" (i.e., producing an answer, which the student has no way of understanding) or resulting in an error that can be baffling. (Similarly, an errant mouse click in an IDE may launch a tool that bewilders the student.)

As an illustrative example, consider this code fragment (taken from a classroom observation by the first author's team): a student early in the semester, tasked with computing a wage, writes the seemingly reasonable

```
wage * hours = salary
```

However, this program is meant to be in C. Not only does this not mean what they expect, in the full C language, it is actually valid to write * on the left side of an assignment, due to pointers—a concept that the student will not confront for many more months. Depending on the compiler this can result in an error about "lvalues", a concept that may not be introduced until much later in the course, making this error message baffling to students.

We can view this problem through the lens of notional machines. Students (at least implicitly) learn notional machines as they progress through computing, and these grow increasingly sophisticated. Ideally, the language they are using matches the notional machine they have been taught until then. The errors we are discussing here are the result of a mismatch, where the implementation provides a much more complex notional machine.

As a result of such observations, many researchers have recognized (Holt & Wortman, 1974; du Boulay, O'Shea & Monk, 1999, p. 268; Findler et al., 2002) the need for sublanguages. Some call for it because they feel that almost any programming language that is useful for constructing real systems probably also contains features that would confuse a beginner. Alternatively, they might want to present all of the language but introduce it gradually, growing the language with the student's learning. In either case, they expect the sublanguage to also provide feedback—such as error messages—at a level appropriate to the student's knowledge. In the C example above, for instance, an early language level would not have pointers at all, the grammar of assignment would be restricted, and the compiler would be expected to report that expressions

can only go on the right of an assignment. In short, the notional machine provided by the implementation would match that being used in education.

There are many ways to design restricted languages. As the example above shows, it could be based on syntactic complexity or the omission of certain features. One particularly useful design criterion—consistent with the principles of this chapter—is to layer the complexity of the notional machine. In DrRacket (Findler et al., 2002), for instance, progressive student levels correspond to increasingly complex notional machines, with concepts like mutation and aliasing—which require a much more detailed notional machine, one that effectively reveals memory layout—appearing only after students have gained familiarity with basic programming techniques.

## 7.2 Errors and Error Messages

Programming environments present error messages in response to several kinds of errors, including syntax errors, type errors, and run-time errors. (In languages with suitable support, there could also be test-failure errors (Felleisen, et al., 2014), algorithm errors (Sudol, 2011; Rivers & Koedinger, 2015), and more). Several studies have attempted to catalog the kinds of errors that students make in different languages and curricula (Hristova et al., 2003; Jackson, Cobb & Carver, 2005; Jadud, 2006; Marceau, Fisler & Krishnamurthi, 2011; Altradmri & Brown, 2015; Tirronen, Uusi-Mäkelä & Isomöttönen, 2015). Each of syntax, semantic, and type errors arise frequently in these studies, with some variation (unsurprisingly) across programming languages (run-time errors are less frequent, as many of these studies worked with only with compilation logs).

Getting students to actually read error messages may be difficult in practice (Denny, Luxton-Reilly & Carpenter; 2104) (though this is not necessarily true of developers (Barik et al., 2017)). Some of the difficulty arguably arises from usability issues with error messages (Traver, 2010; Barik et al., 2014). In languages that support professional-grade programming, for example, error messages default to terminology that experienced programmers understand (such as “lvalue”) that may be beyond novices’ experience with and understanding of the language—once again, reflecting a mismatch between the notional machine of the implementation from that of the education. Some projects attempt to rewrite error messages in beginner-friendly terms (Denny, Luxton-Reilly & Carpenter, 2014), though such rewrites do not necessarily lead to reduced time to resolve errors (Denny, Luxton-Reilly & Carpenter, 2014; Pettit, Homer & Gee, 2017). Other projects have gone further, creating semantically-meaningful subsets of languages (sec. 7.1) that omit constructs or some behaviors of constructs to match what students have learned in various points in a curriculum; these projects produce error messages that are tailored to the language subset and student skill level. These projects use the traditional high-level user-interface for error messages: a textual message anchored to a fragment of code through hyperlinks or highlights. Marceau et al. observed that highlights and anchors may have inconsistent meanings across messages within the same language (Marceau, Fisler & Krishnamurthi, 2011), and proposed guidelines for using multiple highlights as part of consistent principles for connecting error terminology to code.

Lee and Ko's (2011) Gidget project explores a vastly different interface for error messages, one in which the problem of debugging is framed as helping a character find problems in code. This work suggests that an anthropomorphic compiler increases users' willingness to figure out error messages. Such a tool also proposes an interface for presenting notional machines, which in turn leads to issues in debugging. Debugging is beyond the scope of this chapter, but existing surveys provide overviews of work on debugging in an educational context (Fitzgerald et al., 2008).

Overall, creating error messages that engage and support novice, casual, or end-user programmers remains an **open question** in computing education. There are many possible goals for error messages, from aiding in debugging, to identifying misconceptions, to reinforcing terminology about a language. Such work requires attention to many human factors issues, including interface design, cognitive alignment, and user motivation; technical issues are also at play, as a user's errors may reveal misunderstandings of parsing or program execution processes that are not explicit in the user's mental model of how programs run.

## 7.3 Cost Models

A *cost model* tells us how much resource a computation will consume. We usually think of resources as time and space, but many others are also relevant such as energy, network bandwidth, and so on.

Cost models of computation are not routinely covered in the computing-education literature. This may be because so much of the literature is devoted to introductory courses, and the cost of computation—such as big-O models—is sometimes not covered at this level. However, there is some evidence that students do form opinions about the cost of computation before they have been instructed to do so and even when they are explicitly instructed to ignore it (Fisler, Krishnamurthi & Siegmund, 2016).

Cost models are inextricably tied to notional machines, because they depend on the semantics of the language. For instance, consider the *time* cost of evaluating this expression:

```
f(g(x), h(y))
```

In most languages, each of `g(x)` and `h(y)` must first reduce to answers, which are then passed to `f`. Thus, the cost just to start running the body of `f` is the time to evaluate each of the arguments, and a constant for applying `f` itself. In contrast, in a *lazy* semantics (used by languages such as Haskell), neither `g(x)` nor `h(y)` is evaluated right away; rather, they are turned into closures, which take essentially constant time, before `f` begins to evaluate. (The full cost model is more involved.) A reactive model can be even more complicated, since it depends not only on which arguments change, but also on whether the change to the argument results in a change to the result.

A study of cost models thus parallels that of notional machines and recapitulates it. Students form their own models based on brief observations as well as commentary they hear from peers and other sources (e.g., about certain languages being “efficient” or otherwise). They sometimes form flawed models because of a simplistic understanding of the mapping of “lines of code” to operations (sometimes missing that a “line” can be complex and hence take non-constant time (Cooper et al., 2013)), which in turn can impact the way they write code (Fisler, Krishnamurthi & Siegmund, 2016). Understanding student cost models and determining how we can improve their understanding remains an **open question**.

## 7.4 Static Types

A source of some debate in programming education is when one should introduce and use static types. There are natural benefits to types: in addition to the common experience of catching errors early (Tichy & Prechelt, 1998), they also provide an alternate and simpler representation of a program’s behavior, making it possible to study a program and communicate its meaning at a high level. For instance, it should be fairly clear which compositions of the following functions are valid just from their type signatures, without looking at their implementations (indeed, it may be easier from looking at the types than at the implementations):

```
open :: () -> InputPort
close ::  InputPort -> ()
read :: InportPort -> String
```

However, types can also be viewed as problematic for a few reasons. At a time when students may already be struggling with one language they effectively introduce a second one (Tirronen, Uusi-Mäkelä & Isomöttönen, 2015). This language has its own grammar and its own notional machine (a type-checker traverses a program differently than does a compiler or interpreter (Krishnamurthi, 2006)). Furthermore, the checker is essentially invisible except when students make an error—a time when they may already feel nervous or demoralized—and demands to be understood at this fraught time. On the other hand, by catching errors early, it saves significant time and pain later. A thorough study of these tradeoffs does not seem to have been done and **remains open**.

Part of the difficulty in having a discussion about types is that it surely hinges on what notion of types is under consideration. Educators of a certain age may remember the weaknesses of Pascal’s type system, which at least in some quarters earned it a reputation for inflexibility. As one of the creators of SNAP!, Brian Harvey (1993), wrote, "Pascal is part of the same machinery as hall passes, dress codes, advisors' signatures, single-sex dorms, and so on". Part of its inflexibility was due to a lack of parametric polymorphism. In contrast, parametric polymorphism is a natural part of languages like ML and Haskell. In addition, the support these languages offer for type inference lets a programmer write a function as if it has no types, and yet get the benefit of static typing (but at the cost of even more complicated type errors, a long-standing **open problem** in programming languages (Wand, 1986)). Any study of types must therefore take the

richness of the type system into account. Studying the use of type inference in education **remains especially open**.

### 7.5 Non-Standard Programming Models

An enormous amount of attention in computing education has focused on traditional CS1 courses. However, the context of computing is changing rapidly: some CS1 courses are evolving and new courses are being created that deserve just as much attention. Earlier we briefly mentioned the demands of data science (sec 4.2). Another important topic is embedded-systems programming, which is relevant to robotics, the Internet of Things, and so on. Embedded programs are very different from traditional ones in a few ways. First, they follow an event-driven structure with inverted control (Myers, 1991). Second, they are largely non-terminating computations, with no "beginning" and "end". Third, their debugging needs are very different from those of conventional programs. In both topics, notice that the programming models may use quite different notional machines than students are already used to, and these models of computation are far less developed in the literature. Therefore, understanding the needs, difficulties, and misconceptions of students in these courses is **wide open** and an important and pressing task.

Indeed, we will soon need to look even farther, as even the popular press has noticed (Somers, 2017). Program-creation techniques such as program generation from specifications and other formal models (broadly, "synthesis" (Green, 1969)) have already seen industrial success in mission-critical systems and are now reaching maturity in a variety of areas (Gulwani, Polozov, and Singh (2017) offer a useful summary of the state of the art), fueled by advances in various computing technologies from SAT-solving to machine-learning (which introduce entirely new notional machines!). What synthesis techniques consume, however, are not programs in a traditional sense, but rather specifications, which are declarative and require holistic thinking about all possible behaviors of a system—an **open topic** rarely studied in computing education, though there is some knowledge in the broader computing community about the HCI challenges of these techniques (Dix, 2013). This technological advance, with the potential for wide-ranging impacts on all forms of programmers, thus represents entirely new challenges for CEdR.

## 8 Implications Moving Forward

This chapter argues that the conventional focus on "paradigms" is too limited. Today's programming languages embrace features from across the conventional paradigms, and the problems we observe in education tie more to features and particular kinds of notional machines than to the paradigms themselves.

Just as we should embrace going beyond coarse paradigms, we must also stop presenting courses and research with phrases like "we taught the course in language X". This description is too broad. We must discuss the *features* we presented, which problems and examples our course taught and assigned, and which explanations (if not notional machines) we used to describe program behavior to students. Knowing that a course was taught in Python, for

example, does not convey whether students learned imperative programming, list comprehensions, or an integration of the two, yet these distinctions are critical to understanding what kind of programming students are learning.

If we embrace that students should learn to program in different sets of linguistic features over their education, we should better understand how students do and do not evolve their understanding of notional machines and program semantics over time. There has been some work looking at how students transfer knowledge from one language to another (Scholtz & Wiedenbeck, 1990), but this work only rarely extends into upper level curricula (Fisler, Krishnamurthi & Tunnell Wilson, 2017) or looks at how students maintain different mental models of notional machines and language features. This is part of a broader need to extend CEdR beyond first-year courses and novice experiences.

Once we accept that different language features engender different mental models of program evaluation, our studies of how students are learning must try to account for (or at least report) the perspectives that students bring into a new course. For example, a student learning Java after a course in functional programming may well program with different patterns than a student whose prior experience was entirely imperative. Our studies need to consider what programs students had been exposed to previously, and the extent to which certain problems arise more naturally with some language features than with others.

Giving up on “language independent” programming assessments—including those using pseudocode—is another necessary casualty of embracing different models for different sets of language features. Language-“independent” assessments typically depend on a common (subset of a) notional machine, even if the surface syntax changes dramatically; this naturally leaves out all languages that aren’t well explained by that machine, thereby undermining any claims of being “independent” of language. Indeed, assessments that accommodate each of imperative, functional, and reactive models after a first computing course can have little common behavior to draw on (beyond perhaps the behavior of a conditional or interpretation of boolean operators or a Turing Machine). The AP CS Principles framework in the United States (The College Board, n.d.) is an instance of this, boxing curricula into the narrow space of imperative notional machines while claiming to be independent.

The overarching take-away for researchers is to think more deeply about linguistic assumptions and how they interact with pedagogy of prior and current courses. For teachers, we must remember that we choose not just the syntax and IDE in which we will teach; we also choose the pedagogy, problems, and notional machine through which students experience our chosen language. As computing is applied to more and more domains, more end-users become casual programmers, and more languages arise to meet these needs, computing education research gains a wealth of interesting problems to explore that are far more nuanced than covered by our earlier conception of paradigms.

## Acknowledgments

Members of our research group at Brown and WPI— Francisco Castro, Natasha Danas, Justin Pombrio, Sorawee Porncharoenwase, Preston Tunnell Wilson, and John Wrenn—were invaluable in our writing effort. Wrenn, Castro, Danas, and Tunnell Wilson helped conduct literature reviews. Wrenn, Tunnell Wilson, Danas, Pombrio, and Porncharoenwase read closely and provided numerous remarks that greatly improved the quality of this chapter. Castro, Tunnell Wilson, and Wrenn contributed to research that informed this chapter. All participated in numerous stimulating discussions about programming languages and computing education.

We sought input from research colleagues as we scoped this chapter; thanks to Mark Guzdial, Andy Ko, Andrew Luxton-Reilly, Briana Morrison, Guido Rößling, Simon of Newcastle, and Beth Simon for their thoughts on what readers would want from a chapter on this topic. David Weintrop provided multiple rounds of feedback on the details of section 5.3. Colleen Lewis, Andy Ko, Josh Tenenberg, Anthony Robins, Brian Harvey, and Sally Fincher all provided valuable feedback on drafts of this chapter.

Matthias Felleisen and Dan Friedman have influenced our thinking about languages and semantics for decades.

Our work was partially supported by US National Science Foundation grants.

# References


Abelson, Harold, and Sussman, Gerald Jay. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press.

Altadmri, A., and Brown, N. C. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. Pages 522–527 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Armoni, M., Meerbaum-Salant, O., and Ben-Ari, M. 2015. From Scratch to 'real' programming. *Transactions on Computing Education*, **14**(4).

Bailie, Frances, Courtney, Mary, Murray, Keitha, Schiaffino, Robert, and Tuohy, Sylvester. 2003. Objects First - Does It Work? *Journal of Computing Sciences in Small Colleges*, **19**(2), 303–305.

Bainomugisha, Engineer, Carreton, Andoni Lombide, Cutsem, Tom van, Mostinckx, Stijn, and Meuter, Wolfgang de. 2013. A Survey on Reactive Programming. *ACM Computing Surveys*, **45**(4), 52:1–52:34.

Baker, Catherine M., Milne, Lauren R., and Ladner, Richard E. 2015. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems*. ACM.

Barik, Titus, Witschey, Jim, Johnson, Brittany, and Murphy-Hill, Emerson. 2014. Compiler Error Notifications Revisited: An Interaction-first Approach for Helping Developers More Effectively Comprehend and Resolve Error Notifications. Pages 536–539 of: *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion)*. ACM.

Barik, Titus, Smith, Justin, Lubick, Kevin, Holmes, Elisabeth, Feng, Jing, Murphy-Hill, Emerson, and Parnin, Chris. 2017. Do Developers Read Compiler Error Messages? Pages 575–585 of: *International Conference on Software Engineering (ICSE)*. IEEE Press.

Bau, David, Bau, D. Anthony, Dawson, Mathew, and Pickens, C. Sydney. 2015. Pencil code: block code for a text world. Pages 445–448 of: *Proceedings of the International Conference on Interaction Design and Children*.

Begel, A., and Graham, S. L. 2005 (Sept.). Spoken programs. Pages 99–106 of: *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)*.

Ben-Ari, Mordechai. 2010. Objects Never?: Well, Hardly Ever! *Communications of the ACM*, **53**(9), 32–35.

Bennedsen, Jens, and Schulte, Carsten. 2007. What Does "Objects-first" Mean?: An International Study of Teachers' Perceptions of Objects-first. Pages 21–29 of: *Proceedings of the Koli Calling Conference on Computing Education*. Australian Computer Society, Inc.

Bootstrap. 2017 (Jan.). *The Bootstrap Blog–Accessibility (Part 2): Images*. Available online at `http://www.bootstrapworld.org/blog/accessibility/Describing-Images-Screenreaders.shtml`. Last accessed March 28, 2018.

Bootstrap. 2018. *Data Science Curriculum (Spring 2018 edition)*. Available online at `http://www.bootstrapworld.org/materials/spring2018/courses/data-science/english/`. Last accessed March 26 2018.

Brilliant, Susan S., and Wiseman, Timothy R. 1996. The first programming paradigm and language dilemma. Pages 338–342 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Bruce, Kim B. 2005. Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. *SIGCSE Bulletin*, **37**(2), 111–117.

Cartwright, Michelle. 1998. An empirical view of inheritance. *Information and Software Technology*, **40**(14), 795 – 799.

Code.org. *Computer Science in Algebra*. Available online at `https://code.org/curriculum/algebra`. Last accessed March 26, 2018.

College Board, The. *AP Computer Science Principles: Course Overview*. Available online at `https://apstudent.collegeboard.org/apcourse/ap-computer-science-principles`. Last accessed March 26, 2018.

Cooper, Gregory H., Guha, Arjun, Krishnamurthi, Shriram, McCarthy, Jay, and Findler, Robert Bruce. 2013. Teaching Garbage Collection without Implementing Compilers or Interpreters. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Cooper, Keith D., Hall, Mary W., Hood, Robert T., Kennedy, Ken, McKinley, Kathryn S., Mellor-Crummey, John M., Torczon, Linda, and Warren, Scott K. 1993. The ParaScope Parallel Programming Environment. *Proceedings of the IEEE*, **81**(2), 244–263.

Cooper, Stephen, Dann, Wanda, and Pausch, Randy. 2003. Teaching Objects-first in Introductory Computer Science. Pages 191–195 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*. ACM.

Dann, Wanda, Cosgrove, Dennis, Slater, Don, Culyba, Dave, and Cooper, Steve. 2012. Mediated transfer: Alice 3 to Java. Pages 141–146 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

de Raadt, Michael, Watson, Richard, and Toleman, Mark. 2002. Language trends in introductory programming courses. Pages 329–337 of: *Proceedings of Informing Science*.

de Raadt, Michael, Watson, Richard, and Toleman, Mark. 2009. Teaching and Assessing Programming Strategies Explicitly. Pages 45–54 of: *Proceedings of the Australasian Computing Education Conference (ACE)*. Australian Computer Society, Inc.

Dean, Jeffrey, and Ghemawat, Sanjay. 2004. MapReduce: Simplified Data Processing on Large Clusters. In: *Symposium on Operating System Design and Implementation (OSDI)*.

Decker, Rick, and Hirshfield, Stuart. 1994. The top 10 reasons why object-oriented programming can't be taught in CS 1. Pages 51–55 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Denny, P, Luxton-Reilly, A;, and Carpenter, D. 2014. Enhancing syntax error messages appears ineffectual. In: *Proceedings of the SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.

DiSalvo, B. 2012. *Glitch game testers: the design and study of a learning environment for computational production with young African American males*. Ph.D. thesis, Georgia Institute of Technology.

diSessa, Andrea A., and Abelson, Harold. 1986. Boxer: A Reconstructible Computational Medium. *Communications of the ACM*, **29**(9).

Dix, Alan J. 2013. Formal Methods. In: *The Encyclopedia of Human-Computer Interaction, 2nd Ed.* The Interaction Design Foundation.

du Boulay, B. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research*, **2**(1), 57–73.

du Boulay, Benedict, O'Shea, Tim, and Monk, John. 1999. The Black Box Inside the Glass Box. *International Journal of Human-Computer Studies*, **51**(2), 265–277.

Ebrahimi, Alireza. 1994. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies*, **41**, 457–480.

Eckerdal, A., and Thune, M. 2005. Novice Java programmers conceptions of object and class, and variation theory. *SIGCSE Bulletin*, **37**(3), 89–93.

Ehlert, Albrecht, and Schulte, Carsten. 2009. Empirical Comparison of Objects-first and Objects-later. Pages 15–26 of: *Proceedings of the Conference on International Computing Education Research (ICER)*. ACM.

Felleisen, Matthias, and Hieb, Robert. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, **102**, 235–271.

Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, and Krishnamurthi, Shriram. 2001. *How to Design Programs*. MIT Press.

Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, and Krishnamurthi, Shriram. 2009. A Functional I/O System. In: *ACM SIGPLAN International Conference on Functional Programming*.

Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, and Krishnamurthi, Shriram. 2014. *How to Design Programs, 2nd edition*. MIT Press.

Findler, Robert Bruce, Clements, John, Flanagan, Cormac, Flatt, Matthew, Krishnamurthi, Shriram, Steckler, Paul, and Felleisen, Matthias. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, **12**(2), 159–182.

Fisler, Kathi. 2014. The Recurring Rainfall Problem. Pages 35–42 of: *Proceedings of the Conference on International Computing Education Research (ICER)*. ACM.

Fisler, Kathi, Krishnamurthi, Shriram, and Siegmund, Janet. 2016. Modernizing Plan-Composition Studies. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Fisler, Kathi, Krishnamurthi, Shriram, and Tunnell Wilson, Preston. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Fitter, M. 1979. Towards more natural interactive systems. *International Journal of Man-Machine Studies*, **11**(3), 339–350.

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

Flatt, Matthew, Krishnamurthi, Shriram, and Felleisen, Matthias. 1998 (Jan.). Classes and Mixins. Pages 171–183 of: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Fleury, A. E. 1991. Parameter passing: the rules the students construct. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Freeman, Scott, Eddy, Sarah L., McDonough, Miles, Smith, Michelle K., Okoroafor, Nnadozie, Jordt, Hannah, and Wenderotha, Mary Pat. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences of the United States of America*, **111**(23), 8410–8415.

Goldman, Ken, Gross, Paul, Heeren, Cinda, Herman, Geoffrey L., Kaczmarczyk, Lisa, Loui, Michael C., and Zilles, Craig. 2010. Setting the Scope of Concept Inventories for Introductory Computing Subjects. *Transactions on Computing Education*, **10**(2), 5:1–5:29.

Gosling, James, Joy, Bill, Steele, Jr., Guy Lewis, Bracha, Gilad, and Buckley, Alex. 2015. *The Java Language Specification, Java SE 8 Edition*. Oracle America, Inc.

Green, C. Cordell. 1969. Application of Theorem Proving to Problem Solving. Pages 219–240 of: *International Joint Conference on Artificial Intelligence*.

Greening, T. 1999. Emerging constructivist forces in computer science education: shaping a new future. Pages 47–80 of: Greening, T. (ed), *Computer Science Education in the 21st Century*. Springer.

Grover, S., and Basu, S. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Guha, Arjun, Saftoiu, Claudiu, and Krishnamurthi, Shriram. 2010. The Essence of JavaScript. In: *European Conference on Object-Oriented Programming (ECOOP)*.

Gulwani, Sumit, Polozov, Oleksandr, and Singh, Rishabh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages*, **4**(1-2), 1–119.

Gupta, A., Hammer, D., and Redish, E.F. 2010. The case for dynamic models of learners' ontologies in physics. *Journal of the Learning Sciences*, **19**, 285–321.

Guzdial, Mark. 2015. *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. Morgan and Claypool.

Haarslev, V. 1995. Formal semantics of visual languages using spatial reasoning. In: *Proceedings of the IEEE Symposium on Visual Languages*.

Harvey, Brian. 1993 (July). *Discussion forum comment on* `comp.lang.scheme`. Available online at `https://groups.google.com/forum/#!msg/comp.lang.scheme/zZRSDcZdV2M/MZJLiU6gE64J`. Last accessed March 26, 2018.

Harvey, Brian, and Wright, Matthew. 1999. *Simply Scheme: Introducing Computer Science, 2/e*. MIT Press.

Holt, Richard C., and Wortman, David B. 1974. A Sequence of Structured Subsets of PL/I. *SIGCSE Bulletin*, **6**(1), 129–132.

Homer, Michael, and Noble, James. 2014. Combining Tiled and Textual Views of Code. In: *IEEE Working Conference on Software Visualization (VISSOFT)*.

Hristova, Maria, Misra, Ananya, Rutter, Megan, and Mercuri, Rebecca. 2003. Identifying and correcting Java programming errors for introductory computer science students. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Hu, Chenglie. 2004. Rethinking of Teaching Objects-First. *Education and Information Technologies*, **9**(3), 209–218.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, **13**(3), 259–290.

Jackson, J., Cobb, M., and Carver, C. 2005. Identifying top Java errors for novice programmers. In: *Frontiers in Education*.

Jadud, M. C. 2006. Methods and tools for exploring novice compilation behaviour. Pages 73–84 of: *Proceedings of the Conference on International Computing Education Research (ICER)*.

Johnson, L.F. 1995. C in the First Course Considered Harmful. *Communications of the ACM*, **38**(5), 99–101.

Kahn, K. M., and Saraswat, V. A. 1990. *Complete visualizations of concurrent programs and their executions.* Tech. rept. SSL-90-38 [P90-00099]. Xerox Palo Alto Research Centre, Palo Alto, California.

Kay, Alan C. 1993. The Early History of Smalltalk. Pages 69–95 of: *ACM SIGPLAN Conference on History of Programming Languages.* ACM.

Kelleher, C., and Pausch, R. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, **37**(2), 83–137.

Kessel, C. J., and Wickens, C. D. 1982. The transfer of failure-detection skills between monitoring and controlling dynamic systems. *Human Factors*, **24**(1), 49–60.

Ko, A.J., Myers, B.A., and Aung, H. 2004. Six Learning Barriers in End-User Programming Systems. Pages 199–206 of: *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC).*

Kölling, M., Brown, N. C. C., and Altadmri, A. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. Pages 29–38 of: *Workshop in Primary and Secondary Computing Education.*

Krishnamurthi, Shriram. 2006. *Programming Languages: Application and Interpretation.* Available online at `http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/`. Last accessed March 28, 2018.

Krishnamurthi, Shriram. 2008. Teaching Programming Languages in a Post-Linnaean Age. In: *SIGPLAN Workshop on Undergraduate Programming Language Curricula.* Position paper.

Lee, Michael J., and Ko, Andrew J. 2011. Personifying programming tool feedback improves novice programmers' learning. Pages 109–116 of: *Proceedings of the Conference on International Computing Education Research (ICER).*

Lewis, C., Esper, S., Bhattacharyya, V., Fa-Kaji, N., Dominguez, N., and Schlesinger, A. 2014. Children's perceptions of what counts as a programming language. *Journal of Computing Sciences in Colleges*, **29**(4), 123–133.

Lewis, C. M. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. Pages 346–350 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE).* ACM.

Lister, Raymond, Berglund, Anders, Clear, Tony, Bergin, Joe, Garvin-Doxas, Kathy, Hanks, Brian, Hitchner, Lew, Luxton-Reilly, Andrew, Sanders, Kate, Schulte, Carsten, and Whalley, Jacqueline L. 2006. Research Perspectives on the Objects-early Debate. Pages 146–165 of: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education.* ACM.

Ma, Linxiao. 2007. *Investigating and Improving Novice Programmers Mental Models of Programming Concepts.* Ph.D. thesis, University of Strathclyde, Department of Computer & Information Sciences.

Mace, Ronald L., Hardie, Graeme J., and Place, Jaine P. 1991. Accessible Environments: Toward Universal Design. In: Preiser, W. E., Vischer, J. C., and White, E. T. (eds), *Design Intervention: Toward a More Humane Architecture.* Reinhold, NY: Van Nostrand.

Malan, D. J., and Leitner, H. H. 2007. Scratch for budding computer scientists. *SIGCSE Bulletin*, **39**(1), 223–227.

Marceau, Guillaume, Fisler, Kathi, and Krishnamurthi, Shriram. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE).*

Mayer, Richard E., and Moreno, Roxana. 2003. Nine Ways to Reduce Cognitive Load in Multimedia Learning. *Educational Psychologist*, **38**(1), 43–52.

McCauley, Renee, Hanks, Brian, Fitzgerald, Sue, and Murphy, Laurie. 2015. Recursion vs. Iteration: An Empirical Study of Comprehension Revisited. Pages 350–355 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*. ACM.

Meerbaum-Salant, Orni, Armoni, Michal, and Ben-Ari, Mordechai. 2011. Habits of programming in Scratch. In: *Proceedings of the SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.

Miller, Lance A. 1974. Programming by non-programmers. *International Journal of Man Machine Studies*, **6**, 237–260.

Miller, Lance A. 1975. Naive Programmer Problems with Specification of Transfer-of-control. Pages 657–663 of: *Proceedings of the National Computer Conference and Exposition*. ACM.

Miller, Lance A. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, **20**, 184–215.

Miller, Mark S., Dincklage, Daniel Von, Ercegovac, Vuk, and Chin, Brian. 2017. Uncanny Valleys in Declarative Language Design. In: *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Mönig, J., Ohshima, Y., and Maloney, J. 2015. Blocks at your fingertips: Blurring the line between blocks and text in GP. In: *Proceedings of the Blocks and Beyond workshop*.

Muller, Orna, Ginat, David, and Haberman, Bruria. 2007. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. Pages 151–155 of: *Proceedings of the SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM.

Myers, Brad A. 1991 (Nov.). Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. Pages 211–220 of: *ACM Symposium on User Interface Software and Technology*.

Naps, T. L., Rössling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velázquez-Iturbide, J. A. 2003. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, **35**(2), 131–152.

Nelson, Greg L., Xie, Benjamin, and Ko, Andrew J. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In: *Proceedings of the Conference on International Computing Education Research (ICER)*.

Nordström, Marie, and Börstler, Jürgen. 2011. Improving OO Example Programs. *IEEE Transactions on Education*, **54**(Nov.).

Pane, John F. 2002 (May). *A Programming System for Children that is Designed for Usability*. Ph.D. thesis, Carnegie Mellon University, Computer Science Department.

Pane, John F., Ratanamahatana, Chotirat "Ann", and Myers, Brad A. 2001. Studying the Language and Structure in Non-programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies*, **54**(2), 237–264.

Pea, Roy D. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, **2**(1), 25–36.

Pettit, Raymond S., Homer, John, and Gee, Roger. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive. Pages 465–470 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Pirolli, Peter L., and Anderson, John R. 1985. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology/Revue canadienne de psychologie*, **39**(2), 240–272.

Plotkin, Gordon D. 1975. Call-by-name, call-by-value, and the $\lambda$-calculus. *Theoretical Computer Science*, 125–159.

Plum, Thomas. 1977. Fooling the user of a programming language. *Software Practice and Experience*, **7**, 215–221.

Powers, K., Ecott, S., and Hirshfield, L. M. 2007. Through the looking glass: Teaching CS0 with Alice. *SIGCSE Bulletin*, **39**(1), 213–217.

Price, Thomas W., Brown, Neil C.C., Lipovac, Dragan, Barnes, Tiffany, and Kölling, Michael. 2016. Evaluation of a Frame-based Programming Editor. In: *Proceedings of the Conference on International Computing Education Research (ICER)*.

Reges, Stuart. 2006. Back to Basics in CS1 and CS2. Pages 293–297 of: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*. ACM.

Resnick, M., et al. 2009. Scratch: Programming for All. *Communications of the ACM*, **52**(11), 80–87.

Rist, Robert S. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Hum.-Comput. Interact.*, **6**(1), 1–46.

Rivers, K., and Koedinger, K.R. 2015. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 1–28.

Sajaniemi, Jorma, and Kuittinen, Marja. 2008. From Procedures to Objects: A Research Agenda for the Psychology of Object-Oriented Programming Education. *Human Technology*, **4**(1), 75–91.

Savery, John R., and Duffy, Thomas M. 2001 (June). *Problem Based Learning: An instructional model and its constructivist framework*. Tech. rept. 16-01. Indiana University Center for Research on Learning and Technology.

Schanzer, Emmanuel, Krishnamurthi, Shriram, and Fisler, Kathi. 2015. Blocks Versus Text: Ongoing Lessons from Bootstrap. In: *Proceedings of the Blocks and Beyond workshop*.

Scholtz, Jean, and Wiedenbeck, Susan. 1990. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, **2**(1), 51–72.

Schumacher, R. M., and Czerwinski, M. P. 1992. Mental models and the acquisition of expert knowledge. Pages 61–79 of: Hoffman, R. R. (ed), *The Psychology of Expertise: Cognitive Research and Empirical AI*. Springer.

Schwill, A. 1994. Fundamental ideas of computer science. *Bulletin of the European Association for Theoretical Computer Science*, **53**, 274–274.

Seppälä, Otto, Ihantola, Petri, Isohanni, Essi, Sorva, Juha, and Vihavainen, Arto. 2015. Do We Know How Difficult the Rainfall Problem is? Pages 87–96 of: *Proceedings of the Koli Calling Conference on Computing Education*. ACM.

Shinners-Kennedy, D. 2008. The everydayness of threshold concepts: state as an example from computer science. Pages 119–128 of: Land, R., and Meyer, J. H. F. (eds), *Threshold Concepts within the Disciplines*. Sense Publishers.

Simon. 2013. Soloway's Rainfall Problem Has Become Harder. *Learning and Teaching in Computing and Engineering*, 130–135.

Sirkiä, Teemu, and Sorva, Juha. 2012. Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises. Pages 19–28 of: *Proceedings of the Koli Calling Conference on Computing Education*. ACM.

Slotta, J. D., and Chi, M. T. H. 2006. The impact of ontology training on conceptual change: Helping students understand the challenging topics in science. *Cognition and Instruction*, **24**, 261–289.

Smaragdakis, Yannis, and Balatsouras, George. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages*, **2**(1), 1–69.

Soloway, Elliot. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, **29**(9), 850–858.

Soloway, Elliot, Bonar, Jeffrey, and Ehrlich, Kate. 1983. Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, **26**(11), 853–860.

Somers, James. 2017. The Coming Software Apocalypse. *The Atlantic*, Sept.

Sorva, Juha. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph.D. thesis, Aalto University, Department of Computer Science and Engineering.

Sorva, Juha. 2013. Notional Machines and Introductory Programming Education. *Transactions on Computing Education*, **13**(2), 8:1–8:31.

Spohrer, James C., and Soloway, Elliot. 1989. *Simulating Student Programmers*. Morgan Kaufmann Publishers Inc. Pages 543–549.

Stanton, Jim, Goldsmith, Lynn, Adrion, W. Richards, Dunton, Sarah, Hendrickson, Katie A., Peterfreund, Alan, Youngpradit, Pat, Zarch, Rebecca, and Zinth, Jennifer Dounay. 2017 (Mar.). *State of the States Landscape Report: State-Level Policies Supporting Equitable K12 Computer Science Education*. Available online at `http://www.edc.org/sites/default/files/uploads/State-States-Landscape-Report.pdf`. Last accessed March 26, 2018.

Stefik, A., and Gellenbeck, E. 2011. Empirical studies on programming language stimuli. *Software Quality Journal*, **19**(1), 65–99.

Stefik, Andreas, and Gellenbeck, Ed. 2009. Using spoken text to aid debugging: An empirical study. In: *IEEE International Conference on Program Comprehension*.

Stefik, Andreas, and Siebert, Susanna. 2013. An Empirical Investigation into Programming Language Syntax. *Transactions on Computing Education*, **13**(4).

Stefik, Andreas, Hundhausen, Christopher, and Patterson, Robert. 2011. An Empirical Investigation into the Design of Auditory Cues to Enhance Computer Program Comprehension. *International Journal of Human-Computer Studies*, **69**(12), 820–838.

Stefik, Andreas Mikal. 2008. *On the Design of Program Execution Environments for Non-Sighted Computer Programmers*. Ph.D. thesis, Washington State University.

Sudol, Leigh Ann. 2011. *Deepening Students Understanding of Algorithms: Effects of Problem Context and Feedback Regarding Algorithmic Abstraction*. Ph.D. thesis, Carnegie Mellon University.

Tessler, Joe, Beth, Bradley, and Lin, Calvin. 2013. Using Cargo-bot to provide contextualized learning of recursion. In: *Proceedings of the Conference on International Computing Education Research (ICER)*.

Tew, Allison Elliott. 2010 (Dec.). *Assessing fundamental introductory computing concept knowledge in a language independent manner*. Ph.D. thesis, School of Interactive Computing, Georgia Institute of Technology.

Tew, Allison Elliott, McCracken, W. Michael, and Guzdial, Mark. 2005. Impact of alternative introductory courses on programming concept understanding. Pages 25–35 of: *Proceedings of the Conference on International Computing Education Research (ICER)*.

Tichy, Walter F., and Prechelt, Lutz. 1998. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Transactions on Software Engineering*, **24**(4), 302–312.

Tirronen, Ville, Uusi-Mäkelä, Samuel, and Isomöttönen, Ville. 2015. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming*, Aug.

Traver, V. Javier. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, Jan., 1–26.

Tunnell Wilson, Preston, Pombrio, Justin, and Krishnamurthi, Shriram. 2017. Can We Crowdsource Language Design? In: *Proceedings of SPLASH Onward!*

Tunnell Wilson, Preston, Fisler, Kathi, and Krishnamurthi, Shriram. 2017. Student Understanding of Aliasing and Procedure Calls. In: *SPLASH Education Symposium*.

Tunnell Wilson, Preston, Krishnamurthi, Shriram, and Fisler, Kathi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In: *Proceedings of the ACM Symposium on Computer Science Education (SIGCSE)*.

Vasek, Marie. 2012 (May). *Representing expressive types in blocks programming languages*. `https://repository.wellesley.edu/thesiscollection/24/`. Undergraduate thesis, Wellesley College.

Vickers, Paul, and Alty, James L. 2002. When bugs sing. *Interacting with Computers*, Dec., 793–819.

Victor, Bret. 2012 (Sept.). *Learnable Programming: Designing a programming system for understanding programs*. Available online at `http://worrydream.com/#!/LearnableProgramming`. Accessed on March 25, 2018.

Vilner, Tamar, Zur, Ela, and Gal-Ezer, Judith. 2007. Fundamental concepts of CS1: procedural vs. object oriented paradigm–a case study. *SIGCSE Bulletin*, **39**(3), 171–175.

Wand, Mitchell. 1986. Finding the Source of Type Errors. Pages 38–43 of: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Weintrop, D., and Wilensky, U. 2015a. To Block or not to Block, That is the Question: Students Perceptions of Blocks-based Programming. Pages 199–208 of: *Proceedings of the International Conference on Interaction Design and Children*.

Weintrop, D., and Wilensky, U. 2015b. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In: *Proceedings of the Conference on International Computing Education Research (ICER)*.

Weintrop, D., and Wilensky, U. 2017a. Comparing Blocks-based and Text-based Programming in High School Computer Science Classrooms. *Transactions on Computing Education*, **18**(1).

Weintrop, David, and Wilensky, Uri. 2017b. Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments. In: *Proceedings of the International Conference on Interaction Design and Children*.

Wiedenbeck, Susan, and Ramalingam, Vennila. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, **51**(1), 71 – 87.

Wiedenbeck, Susan, Ramalingam, Vennila, Sarasamma, Suseela, and Corritore, Cynthia L. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, **11**(3), 255 – 282.