Purpose-first Programming: A Programming Learning Approach for Learners Who Care Most About What Code Achieves

Kathryn Cunningham
School of Information
Ann Arbor, Michigan, USA
kicunn@umich.edu

ABSTRACT

Becoming "a programmer" is associated with gaining a deep understanding of programming language semantics. However, as more people learn to program for more reasons than creating software, their learning needs differ. In particular, end-user programmers and conversational programmers often care about code's purpose, but don't wish to engage with the low-level details of precisely how code executes. I propose the creation of scaffolding that allows these learners to interact with code in an authentic way, highlighting code's purpose while providing support that avoids the need for low-level tracing knowledge. This scaffolding builds on theories of programming plans.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; *CS1.*

KEYWORDS

CS1; structure behavior function; tracing; programming plans

ACM Reference Format:
Kathryn Cunningham. 2020. Purpose-first Programming: A Programming Learning Approach for Learners Who Care Most About What Code Achieves. In *2020 International Computing Education Research Conference (ICER '20), August 10–12, 2020, Virtual Event, New Zealand.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3372782.3407102

1 INTRODUCTION

Alan Perlis said, *"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy"* [4]. Unfortunately, much of an introductory programming course can feel like the "Turing tar-pit." CS1 courses focus on building generalizable programming knowledge by focusing on a language's syntax and semantics. Assignments often involve "code tracing" problems, where students perform close tracking of code's execution, typically in the context of 'toy' problems [6]. Code tracing has empirical evidence for helping novice programmers solve problems like debugging and explaining code in a natural language [3]. As a result, "reading-first" approaches propose that code tracing should be taught early to novice programmers, even before they have the opportunity to write code [3, 7].

ICER '20, August 10–12, 2020, Virtual Event, New Zealand

ACM ISBN 978-1-4503-7092-9/20/08.
https://doi.org/10.1145/3372782.3407102

However, many learners do not perform code tracing, even in situations when it is helpful for others [1]. When studying code tracing behavior using "sketched traces", I found that learners sketched more frequently on certain types of problems, and were least likely to succeed when they started tracing but didn't finish [1]. Code tracing activity isn't a simple panacea for problem-solving.

In my thesis, I investigate the reasons that novice programmers don't trace code and use the results to inform the development of a new programming learning approach: *purpose-first programming*.

2 LITERATURE REVIEW

How deeply do programmers need to understand a program? The Structure Behavior Function framework [2] offers some insight. This framework proposes that to understand how a designed thing (like a piece of code) works, one must have some understanding of that code's behavior. Understanding involves looking at code's structure and determining what its purpose or function is, using the code's behavior to figure it out.

By mapping programming learning hierarchies (e.g. [7]) onto the Structure Behavior Function framework, it becomes clear that these hierarchies typically emphasize *code tracing* as their representation of code behavior. Code tracing is relatively low-level, involving close tracking of changes in memory values over time [6]. Is this the only way to conceptualize code behavior?

Soloway and his colleagues provided evidence that both novice and expert programmers have schemas that match commonly used code patterns, which they termed *programming plans*. Programming plans are small program fragments that achieve a goal, like selecting values from a list that match a certain criteria [5]. Programming plans offer a possibility to connect code structure and function more directly than through low-level tracing.

3 RESEARCH GOALS & PROGRESS

3.1 RQ1: Why do novices avoid code tracing?

In my first study, I interviewed 13 CS1 students retrospectively about their decisions to sketch code traces on code prediction problems. I found that when novices didn't sketch, they often reported that they'd identified the overall goal that the code achieved, and didn't need to perform the cognitively taxing work of tracing. When students did sketch, I found that their sketching choices were driven by a search for a program's patterns, an attempt to create organizational structure among intermediate values, and the tracking of prior steps and results. The discovery of a pattern could lead to the halting if a trace midway. Whether they traced or not, novices were searching for the functionality and purpose of code, rather than merely tracing its behavior.

In a second study, I performed thinkaloud interviews about a code tracing task with 12 undergraduate and graduate novice programmers. Unexpectedly, some participants described ways that their identifies related to their decision to avoid tracing code. In two case studies, I explored identities that relate to choices about code tracing: *I'm not a computer* and *I'm not a programmer*. Learners related these self-beliefs to a low expectation of success on code tracing, because they did not have the ability to notice code details or chose not to remember them. They also expressed a low value for code tracing, because it took a lot of effort, was not enjoyable, and did not appear relevant to their self-image and goals. In both case studies, participants defined themselves at a distance from people who are thinking deeply about how code works. However, from this distance, they saw themselves as someone who uses programming, but relies on the machine to work through the details.

In combination, these studies show why tracing activities can create a "perfect storm" for discouraging learners from completing them: they require high cognitive load, leading to a low expectation of success, while also being disconnected from meaningful code, resulting in low value for the task. Instead of carefully stepping through the actions of code, learners were interested in identifying the *purpose* of code and using code to achieve their own goals. However, they may need support to discern code's goals accurately.

3.2 RQ2: How can we scaffold code writing and reading for those who avoid tracing?

If novices can quickly and easily create or understand authentic code without the need for deep knowledge of semantics, it may result in both a greater expectancy of success and a higher task value, motivating engagement with this programming activity. Many learners may not care about exactly how a programming language works, but they do care about code's *purpose*—what code can achieve for them. I draw on cognitive science and theories of motivation to describe a "purpose-first" programming pedagogy.

Programming plans are an ideal foundation for purpose-first programming because they associate a piece of code with the purpose it serves [5]. I plan to develop a learning experience that scaffolds understanding of code behavior, using programming plans.

3.2.1 Operationalizing the definition of programming plans. To be useful in a curriculum, the definition of a programming plan must expand beyond the simple association of a goal with code.

- I will define a plan as fixed code and *"slots"* that can contain objects or code. While prior plan editors allowed only numbers or strings to fill a slot, I will allow slots to contain not only literal values, but also other code.
- I will highlight the meaning of variables by describing the content of slots with a *domain-specific meaning*. Domain concepts connect code to action in the real world.
- I will add *subgoals* to plans to guide plan integration and tracing. Subgoals provide a smaller unit of cohesive code, and a subgoal label clarifies the contribution to the plan's purpose. By tracing the input and output to each subgoal, learners can trace purpose-first code at a higher level of abstraction.

3.2.2 Validating a set of plans for coverage and authenticity. I will develop an initial set of plans manually, based on patterns observed in code in the domain of web scraping. I will generate 5-8 plans, then, using feedback from domain experts, I will iteratively refine the aspects of my plan definition as well as the set of plans. To gather feedback from domain experts, I will hold an interview session with data scientists from the University of Michigan School of Information who perform web scraping in their work, and ask whether the identified plans are truly useful in the domain.

3.2.3 Developing scaffolded activities. I will create a web-based tool where learners can learn about web scraping using plan-based activities, such as filling in missing code pieces for a particular plan, combining plans to achieve a goal, and explaining what a plan does in natural language.

3.3 RQ4: Are purpose-first scaffolds effective?

I will evaluate learners' experiences with purpose-first programming with a qualitative study focusing on (1) expectancy of success, (2) task value, and (3) usability of the curriculum. I will run a study where 12 novices in an introductory data-focused programming course with no prior experience in web scraping complete scaffolded learning and assessment activities in the online tool. Before engaging with the curriculum, I will survey participants about their interest in web scraping and their attitudes toward code tracing. During their completion of the problems, I will observe and catalog any usability issues with the purpose-first activities. After completion of the activities, I will interview students about their perceptions of the authenticity of these tasks for their future goals, to understand task value, and their belief that they could accurately complete future purpose-first activities, to understand expectancy of success.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGC-1148903.

REFERENCES

[1] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, 164–172.

[2] Ashok K Goel, Andrés Gómez de Silva Garza, Nathalie Grué, J William Murdock, Margaret M Recker, and T Govindaraj. 1996. Towards design learning environments — I: Exploring how devices work. (1996).

[3] Raymond Lister. 2011. Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11)*. Australian Computer Society, Inc., AUS, 9–18.

[4] Alan J Perlis. 1982. Special feature: Epigrams on programming. *ACM Sigplan Notices* 17, 9 (1982), 7–13.

[5] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (Sept. 1984), 595–609.

[6] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages.

[7] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.