

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

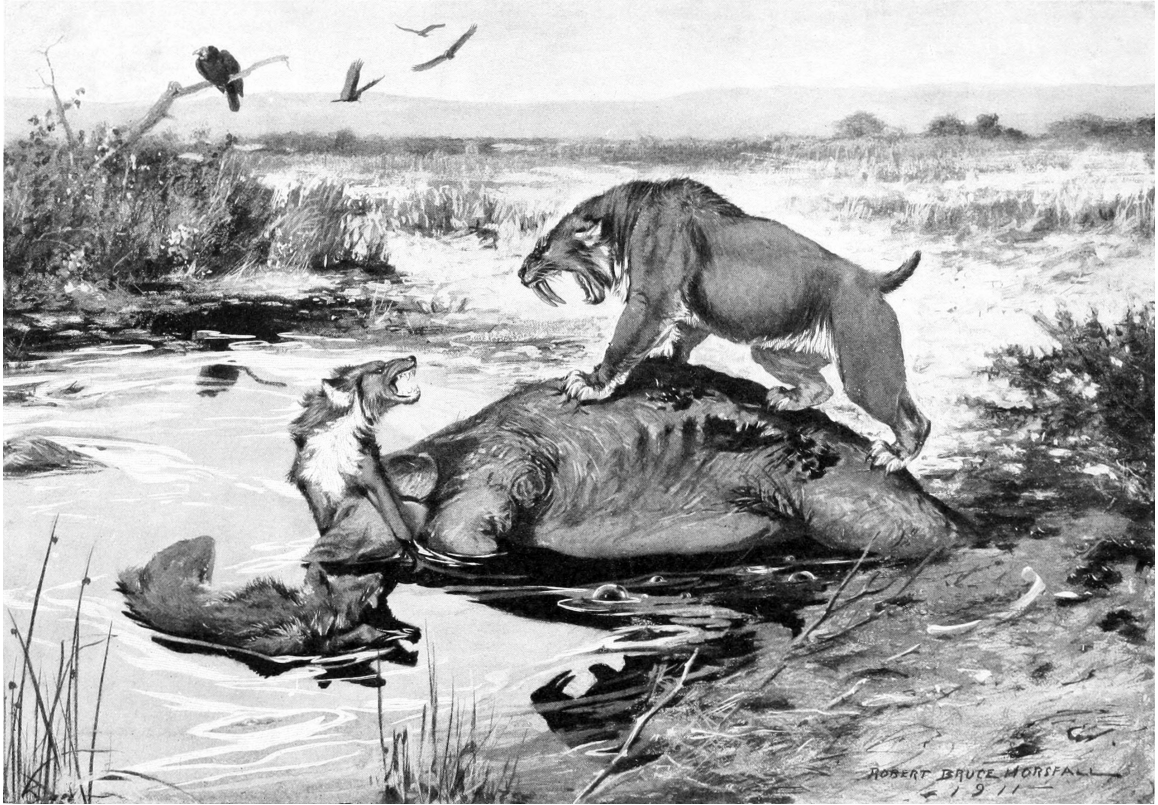
by

Kathryn Irene Cunningham

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information)
in The University of Michigan
2021

Doctoral Committee:

Assistant Professor Barbara Ericson, Co-Chair
Professor Mark Guzdial, Co-Chair
Associate Professor Chris Quintana
Research Professor Stephanie Teasley



Smilodon californicus and *Canis dirus* fight over a *Mammuthus columbi* carcass in the La Brea Tar Pits.

Artist: Robert Bruce Horsfall

Source: William Berryman Scott, A history of land mammals in the western hemisphere, New York, MacMillan Publishing Company, 1913. Frontispiece.

Kathryn Irene Cunningham
kicunn@umich.edu
ORCID iD: 0000-0002-9702-2796

© Kathryn Irene Cunningham 2021

ACKNOWLEDGEMENTS

I have been very fortunate to work on my doctorate with support from my research community, my friends, and my family. Thanks to everyone who encouraged me to write this thesis, and who helped me see it through to the end.

Mark Guzdial and Barbara Ericson provided invaluable guidance and knowledge as they advised me through my doctorate and into the next step in my career. I am so grateful for not only their support for my research, but also their emotional support as I weathered the challenges of moving from Georgia Tech to Michigan.

Many times, my peers were my greatest motivators. My classmates at Georgia Tech encouraged me to live up to their example as rock star researchers, and the community of computing education graduate students constantly renewed my passion for our field.

The undergraduate researchers I've worked with likely didn't know how helpful their excitement and enthusiasm was to my progress. My former students at Hartnell Community College and California State Monterey Bay are often in my thoughts, and I thank them for reminding me about the life-changing impact of quality computer science education.

My friends have been there for me when I needed it most, with encouraging words, letters, and video calls. Special thanks to Rebecca Krosnick for understanding both my love of baseball and the struggle of maintaining self-confidence in graduate school, and to Eshwar Chandrasekharan for his listening ear, kind words, and strength.

Thanks to my parents and my brother for their endless support and love.

Thanks to the National Science foundation for funding my research and having confidence in my future as a researcher. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1256260.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	ii
LIST OF FIGURES	ix
LIST OF TABLES	xi
ABSTRACT	xii
CHAPTER	
I. Overview of the Thesis	1
1.1 Background: Code’s structure, behavior, and function	2
1.2 Understanding the challenge: Why novices avoid code tracing	4
1.2.1 Novice rationales for sketching and tracing, and how they try to avoid it	4
1.2.2 “I’m not a computer”: How identity informs value and expectancy during a programming activity	6
1.3 Meeting the challenge: Purpose-first programming	7
1.3.1 How it’s designed	8
1.3.2 Why it might work	8
1.3.3 How I evaluated it and what I found	8
1.4 Implications for programming learning	9
II. Using the Structure Behavior Function Framework to Understand Programming Learning	11
2.1 The Structure Behavior Function framework	12
2.1.1 What is the Structure Behavior Function framework?	12
2.1.2 How do structure, behavior, and function interact?	13
2.1.3 Where did SBF originate?	14
2.1.4 What do we know about SBF in learning environ- ments?	18

2.2	What are Structure, Function, and Behavior in the context of programming?	20
2.2.1	Structure of a program	20
2.2.2	Behavior of a program	24
2.2.3	Function of a program	27
2.2.4	Connections between structure, function, and behavior.	29
2.3	Three theories of programming instruction through an SBF lens	31
2.3.1	The “Neo-Piagetian” hierarchy of programming skills	31
2.3.2	Xie et al.’s theory of programming instruction	33
2.3.3	Schulte’s Block Model	37
2.3.4	Discussion	40
2.4	Conclusion	41

III. Novice Rationales for Sketching and Tracing, and How They Try to Avoid It 43

3.1	Introduction	43
3.2	Background	45
3.2.1	Tracing and other ways novices read code	45
3.2.2	When and why do students sketch?	46
3.2.3	What external representations are common in novice programming?	47
3.3	Method	48
3.3.1	Class observations	49
3.3.2	Interviews	49
3.4	Why sketch (or not)?	50
3.4.1	Goal and pattern recognition	50
3.4.2	Anticipated cognitive load	51
3.4.3	Problem-solving progression: From goal search to tracing (and re-tracing)	52
3.5	Why not sketch like the instructor?	53
3.5.1	Seen as unnecessary and time-consuming	54
3.5.2	Visual details seem distant from code	54
3.5.3	Boxes are reserved for another purpose	55
3.6	What do novice sketches include?	55
3.6.1	Organizing and structuring of traces	55
3.6.2	Persistence of past values and calculations	56
3.6.3	Anchoring with visible values and structures	57
3.7	Discussion	58
3.7.1	Search for goals and patterns is primary	58
3.7.2	Variables are treated differently based on context	59
3.7.3	Past values are retained	60

3.7.4	Variables are un-boxed	60
3.8	Limitations and threats to validity	60
3.9	Conclusion	61
IV.	“I’m not a computer”: How Identity Informs Value and Expectancy During a Programming Activity	63
4.1	Introduction	64
4.2	Background	65
4.2.1	Identity, programming, and applications	65
4.2.2	The Eccles expectancy-value model of achievement choice	66
4.2.3	Tracing code to solve problems	67
4.3	Method	68
4.3.1	Task	68
4.3.2	Problems	68
4.3.3	Participants	69
4.3.4	Interview Protocol	69
4.3.5	Analysis	70
4.4	Case Studies	71
4.4.1	Charles: I’m not a computer	71
4.4.2	Luke: I’m not a programmer	73
4.5	Discussion	76
4.5.1	Previous achievement-related experiences, Inter- pretations, and Affective reactions and memories	77
4.5.2	Goals and general self schemata	78
4.5.3	Activity-specific ability, self-concept, and expecta- tions for success	78
4.5.4	Subjective task value: Interest-enjoyment value	78
4.5.5	Subjective task value: Attainment value	79
4.5.6	Subjective task value: Utility value	79
4.5.7	Subjective task value: Relative cost	79
4.6	Conclusion	80
V.	Defining, Building, and Evaluating Purpose-First Programming	82
5.1	Introduction	82
5.1.1	Summary of contributions	84
5.2	Motivation for the approach	86
5.2.1	Conversational programmers and end-user program- mers want to understand the purpose of complex code, but also want to avoid detailed semantics	86
5.2.2	Programming tools for non-developers often avoid industry- standard code, so they don’t provide dis- ciplinary authenticity	87

5.2.3	Plans may be a more motivating way for conversational programmers to think about code	88
5.2.4	Other systems have provided plan- or example-based support	89
5.3	Formative study: Investigating the responses of novice programmers to purpose-oriented assistance	90
5.3.1	Focus group results	90
5.3.2	Survey results	92
5.3.3	Conclusions from the formative study	95
5.4	Defining Purpose-first programming	96
5.4.1	Identifying authentic, domain-specific programming plans	96
5.4.2	Expanding the definition of a programming plan to serve instructional needs	97
5.4.3	Providing "glass-box" scaffolding to support learners as they work with plans	99
5.5	Designing the purpose-first programming proof-of-concept curriculum	100
5.5.1	Building a set of plans	100
5.5.2	Creating activities	102
5.5.3	Selecting a platform	103
5.5.4	Designing purpose-first support	103
5.5.5	How this prototype meets the design goals	105
5.6	Method	108
5.6.1	Study Design	108
5.6.2	Recruitment and participants	109
5.7	Evaluation of learners' problem-solving	111
5.7.1	Learners were able to complete scaffolded writing, debugging, and code explanation tasks	111
5.7.2	Participants used purpose-first scaffolds to apply plan knowledge and complete tasks	112
5.7.3	Discussion	121
5.8	Evaluation of learners' motivation	123
5.8.1	Analysis approach	123
5.8.2	Participants were motivated to learn with purpose-first programming in the future	124
5.8.3	Learners perceived the purpose-first programming curriculum as having low cognitive load	124
5.8.4	Participants felt success and enjoyment, which came from understanding and completing problems . . .	128
5.8.5	Participants felt that purpose-first programming was for beginners and those who need extra help .	130
5.8.6	Participants believed purpose-first programming gave them conceptual, high-level knowledge	133

5.8.7	Participants found curricular content realistic and applicable	135
5.8.8	Code tracing visualizations are unhelpful if confusing, but useful for deep knowledge if understandable	136
5.8.9	Discussion	138
5.9	Implications for future curriculum design	141
5.9.1	There are opportunities to streamline activities and reduce user error	141
5.9.2	Clear indication of success can contribute to motivation	141
5.9.3	Fading of scaffolding could allow purpose-first programming to be more broadly used	142
5.10	Conclusion	143
VI. Summing Up and Looking Forward		144
6.1	Contributions of this thesis	144
6.1.1	Novices don't trace code because it's cognitively challenging and has low value	144
6.1.2	Learning domain-specific programming plans can motivate both conversational and end-user programmers	145
6.1.3	Considering <i>both</i> cognition and value can lead to effective approaches	146
6.2	Future work	148
6.2.1	What learner characteristics are correlated with a benefit from purpose-first programming?	148
6.2.2	How can purpose-first programming activities fit into an instructional sequence?	148
6.2.3	What technology can support development of purpose-first programming curricula at scale?	149
BIBLIOGRAPHY		152

LIST OF FIGURES

Figure

1.1	A complete sketched trace.	5
2.1	An illustration of the SBF framework	14
2.2	The rainfall problem with syntactical substructures highlighted (left) and plan substructures highlighted (right).	23
2.3	The abilities of learners in Lister’s Neo-piagetian stages, mapped to the SBF model	33
2.4	Xie et al.’s four skills from their theory of instruction [163]. ©Benjamin Xie	35
2.5	Xie et al.’s instructional stages, mapped to the SBF framework . . .	37
3.1	Variable visualizations in program animation tools	48
3.2	An in-class sketch by the instructor.	53
3.3	Tracing organized by loop indices	56
3.4	Tracing organized around a list	57
3.5	Persistence of past values	58
3.6	Organizing tracing around code structure	59
4.1	A participant’s work on Problem 1, during the tracing stage.	68
4.2	Charles’ self-narrative mapped onto relevant aspects of the Eccles Expectancy-Value Model of Achievement Choice [43].	74
4.3	Luke’s self-narrative mapped onto relevant aspects of the Eccles Expectancy-Value Model of Achievement Choice [43].	76
4.4	Relevant components of the Eccles expectancy-value model of achievement choice (modified from Eccles [43]). Cultural Milieu, Socializer’s Beliefs and Behaviors, Stable Child Characteristics, and Perception of Stereotypes and Socializer’s Beliefs are not represented. .	77
5.1	The development of a purpose-first programming module	84
5.2	Survey respondents were asked to rank and reflect on the usefulness of these code writing activities with different levels of purpose-oriented scaffolding. The directions for all activities were: “Complete the code that achieves the goal”.	92

5.3	An example plan from the domain of web scraping. This plan achieves an authentic goal in its domain, and consists of multiple subgoals and slots. The slots define the space of relevant domain knowledge needed to work with this plan.	97
5.4	Plan usage across the curriculum. Activities use novel plan combinations not seen in instructional examples.	103
5.5	Slot highlighting, plan goals, and subgoals assist learners as they debug this code.	104
5.6	Practice activities contain subgoal label scaffolding, and focus only on knowledge about plan slots.	105
5.7	All plan instruction is situated in examples. Learners first view and run a complete program example, then learn about how each plan contributes to the full program.	106
5.8	Writing code takes place in stages. Learners first assemble plans, and then fill in plan slots.	107
5.9	Overview of the study design	109
5.10	Traces of participants' activities when solving code Writing part 3 (fill plan slots)	118
5.11	Selected quotes from P9's thinkaloud mapped onto the actions she took while solving Writing Activity 3 (fill plan slots).	121
5.12	Selected quotes from P5's thinkaloud mapped onto the actions she took while solving Writing Activity 3 (fill plan slots).	122
5.13	The themes mapped onto the Eccles Expectancy-Value Model of Achievement Choice [42].	139

LIST OF TABLES

Table

2.1	The Block Model, adapted from [20].	38
3.1	Code plans appearing in code reading questions.	49
5.1	Difference in attitudes about the highly scaffolded code writing problem (Figure 5.2a) between conversational programmers and non-conversational programmers. Attitudes were drawn from focus group quotes. Survey respondents were asked to rate their agreement on a 7-point Likert scale, Strongly Disagree - Strongly Agree.	93
5.2	Participants' success and time to completion on scaffolded activities (n=9).	112
5.3	Participants' mentions of goals and subgoals during their thinkaloud on each activity.	115
5.4	Examples of different participants focusing on subgoals or on code at the same step in an activity. Mentions of subgoals are bolded, and mentions of code are underlined. All quotes are from conversational programmers.	117
5.5	Participants' visits to plan reference pages in Writing Activity 3. "Most relevant" plans are plans where the participant's code contained an error or was incomplete at the time of the page visit. . . .	118
5.6	Self-reported cognitive load on scaffolded activities (n=9). Cognitive load is measured on a 9-point scale (Very very low mental effort (1) - Very very high mental effort (9) [111]).	127

ABSTRACT

Introductory programming courses typically 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. "Reading-first" approaches propose that code tracing should be taught early to novice programmers, even before they have the opportunity to write code.

However, many learners do not perform code tracing, even in situations when it is helpful for other students. To learn more, I talked to novice programmers about their decisions to trace and not trace code. Through these studies, I identified both cognitive and affective factors related to learners' motivation to trace. My research found that 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.

These findings suggest that a new learning approach, where novices quickly and easily create or understand useful code without the need for deep knowledge of semantics, may lead to higher engagement. Many learners may not care about exactly how a programming language works, but they do care about what code can achieve for them.

I drew on cognitive science and theories of motivation to describe a "purpose-

first” programming pedagogy that supports novices in learning common code patterns in a particular domain. I developed a proof-of-concept “purpose-first” programming curriculum using this method and evaluated it with non-major novice programmers who had a variety of future goals.

Participants were able to complete scaffolded code writing, debugging, and explanation activities in a new domain (web scraping with BeautifulSoup) after a half hour of instruction. An analysis of the participants’ thinkalouds provided evidence the learners were thinking in terms of the patterns and goals that they learned with in the purpose-first curriculum.

Overall, I found that these novices were motivated to continue learning with purpose-first programming. I found that these novices felt successful during purpose-first programming because they could understand and complete tasks. Novices perceived a lower cognitive load on purpose-first programming activities than many other typical learning activities, because, in their view, plans helped them apply knowledge and focus only on the most relevant information. Participants felt that what they were learning was applicable, and that the curriculum provided conceptual, high-level knowledge. For some participants, particularly conversational programmers who didn’t plan to program in their careers, this information was sufficient for their needs. Other participants felt that purpose-first programming was a starting point, from which they could move forward to gain a deeper understanding of how code works.

CHAPTER I

Overview of the Thesis

Alan Perlis said, “Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy” [115]. Unfortunately, much of an introductory programming (CS1) 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 [2]. Assignments often involve “code tracing” problems, where students perform close tracking of code’s execution, typically in the context of ‘toy’ problems [140]. Theoretically, code tracing should help students develop a key competency: an understanding of the way the computer executes code, known as the “notional machine” [38, 140]. Code tracing has empirical evidence for helping novice programmers solve problems like debugging [103] and explaining code in a natural language [86]. 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 [163, 110, 86].

However, many learners do not perform code tracing, even in situations when it is helpful for other students [90]. When studying code tracing behavior using “sketched traces”, I found that learners sketched more frequently on certain problems than others, and were least likely to succeed when they started tracing, but didn’t complete a trace [33]. Code tracing activity doesn’t seem like a simple

panacea for problem-solving.

In order to learn more, I talked to novice programmers about their decisions to trace and not trace code. Through these studies, I identified both cognitive and affective factors related to learners' motivation to trace. My research found that tracing activities can create a "perfect storm" for discouraging learners from completing them: they require high cognitive load [34], leading to a low expectation of success [32], while also being disconnected from meaningful code, resulting in low value for the task [32].

These findings suggest that a new learning approach, where novices quickly and easily create or understand useful code without the need for deep knowledge of semantics, may lead to higher engagement. Many learners may not care about exactly *how* a programming language works, but they do care about *what* code can achieve for them.

I draw on cognitive science and theories of motivation to describe a "purpose-first" programming pedagogy that supports novices in learning common code patterns in a particular domain. I developed a proof-of-concept "purpose-first" programming curriculum using this method and evaluated it with non-major novice programmers who had a variety of future goals. I found that these novice programmers were able to use the pattern-based approach to solve problems. I also found that novices were motivated to learn with this method in the future, because they felt that they were successful on purpose-first programming problems and that the content was valuable in helping them achieve their goals.

1.1 Background: Code's structure, behavior, and function

In Chapter II, I introduce the Structure Behavior Function (SBF) framework [35, 51, 55], a general model that describes the cognition necessary to understand and create artifacts that serve a purpose. In programming education research, and

in tracing research in particular (e.g.[88]), many code examples are collections of syntax structures that don't achieve any "real-world" goal. As one of my participants said, the only purpose of such problems is "teaching people Python" (Chapter IV). By contrast, during purpose-first programming, novices only learn code patterns that achieve goals in a domain of professional practice (e.g. web scraping). Theory directs researchers' attention by bringing certain aspects of what they study into sharper focus than others [65]. By thinking about code understanding and creation using the SBF framework, I assured code's purpose was always in focus.

I used the SBF framework to organize relevant background literature about novice program comprehension and creation. The SBF framework proposes that to understand or create a designed artifact, one must utilize an understanding of the artifact's *behavior*. I highlight a variety of ways that we can define code behavior, from programming language semantics to more abstracted approaches like programming plans [134].

I map existing programming learning hierarchies [163, 86, 129] onto the Structure Behavior Function framework, in order to make comparisons between them. I find that these hierarchies typically define code behavior as the operation of programming language semantics (e.g. the branching of a selection statement, or the creation of a variable). These hierarchies only focus on "higher-level" understandings of code behavior, like recognizing a pattern of code, at later stages of the hierarchy. In these progressions, it will take students a long time to interact with meaningful code.

1.2 Understanding the challenge: Why novices avoid code tracing

1.2.1 Novice rationales for sketching and tracing, and how they try to avoid it

In Chapter III, I identify cognitive reasons that novice programmers don't trace code, or stop tracing partway through. When novices don't trace, they often report that they have identified the goal that the code achieves, without needing to carefully track every mechanism of how the code operates. When novices start but don't complete a code trace in its entirety, they claim they found a pattern and don't need to finish tracing. When these learners could not determine a goal or pattern, they resorted to detailed and complete code tracing. Overall, novices use a problem-solving strategy that prioritizes a search for the *functionality* of code, rather than merely tracing its *behavior*.

1.2.1.1 Summary of methods and findings:

In this study, I interviewed 13 introductory programming (CS1) students retrospectively about their decisions to draw out code traces on paper while solving code prediction problems on a recent Python programming exam. The "sketches" on students' scratch sheets are an artifact that captures their code tracing activity [88, 33] (see Figure 1.1). Using each student's scratch sheet to ground the interview, I performed a retrospective artifact walkthrough where I asked students about how they made choices to trace or not trace, and why they chose to draw their traces in a particular way.

I found that rather than immediate use of a tracing technique, interviewees typically described an initial search for code's meaning, then a fall-back to tracing when no discernible goal or pattern was found. When they stopped tracing partway, these learners reported that they had discovered a pattern or goal, and didn't need to finish tracing.

$for\ i\ in\ range(1,3)$
 $value = numList[i]$
 $value = 20$
 $sum += 20$

$1, 2.$
 $value = numList[2]$
 $value = 30$
 $sum = 50.$

$if\ (2 - 1 + 1) >= 1$
 $(return\ 50 / (2 - 1 + 1))$
 $50 / 2 = 25$

Figure 1.1: A complete sketched trace.

Even while sketching and tracing, novices' code traces were organized in ways driven by the search for a program's patterns and goals. Students de-prioritized certain variables based on their function (e.g. illustrating a loop variable differently than a variable that holds a sum). Students preferred to keep prior values around for reference, in order to better track patterns.

These findings highlight a key difference between humans and compilers: while a computer must parse and execute code token by token and line-by-line, human learners can infer patterns across time and identify global goals. Even on a type of problem designed to be solvable by code tracing alone, novices tried to infer patterns and identify goals. This approach can save time and cognitive load, however, the novices were not always successful with this strategy. Scaffolding that helps learners better identify goals and patterns in code may not only align with novices' preferred problem-solving strategies, but also help them be more successful.

1.2.2 “I’m not a computer”: How identity informs value and expectancy during a programming activity

In Chapter IV, I explore ways some learners connect their self-identity to their decision to not trace code. I found that some struggling novice programmers described code tracing as not only cognitively complex, but also in opposition to their self-beliefs. One participant described himself as *not a computer*, and therefore unfit to execute code like the computer does. Another described himself as *not a programmer*, and did not value an activity that was only for learning about how code works. While both participants valued what they could create with code, neither valued code tracing. Alternative activities that focus on code’s purpose might allow students with these identities to build skills in a way that aligns with their self-beliefs.

1.2.2.1 Summary of methods and findings:

I performed retrospective and think-aloud sessions about code tracing activity in Python with 12 undergraduate and graduate novice programmers from an Information major. During the interviews, learners not only described their cognition, but also their judgments about code tracing tasks. Two novices refused to continue with code tracing activities, and provided explanations for avoiding code tracing that were connected to self-beliefs and goals. I used Values Coding to identify these learners’ values, attitudes, and beliefs about code tracing, and then mapped them onto elements of the Eccles Expectancy-Value Model of Achievement Choice [41] to understand how the decision to not trace code is related to identity.

I found two self-beliefs that relate to choices about code tracing: *I’m not a computer* and *I’m not a programmer*. In my case studies, learners related these self-schemata 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. Code tracing was seen as an academic exercise for learning a programming language rather than something useful for creating code in the "real world". In both case studies, participants define themselves at a distance from people who are thinking deeply about how code works. However, from this distance, both participants see themselves as someone who uses programming, while relying on the machine to work through the details.

These learners are using code to achieve their goals, but they reject code tracing, a common activity in programming classrooms. To meet this type of programming learner where they are, I propose an exploration of programming learning activities that are purpose-oriented, contextualized, and authentic. If code tracing can be performed in a way that is less cognitively demanding and more directly related to code's purpose, it may result in a higher expectancy of success, higher task value, and ultimately higher motivation.

1.3 Meeting the challenge: Purpose-first programming

In Chapter V, I propose *purpose-first programming*, a new approach to programming learning that emphasizes code's *purpose*, rather than programming language semantics. The goal of this approach is to motivate learners to engage in the creation and understanding of authentic programs from a domain related to their interest, without the need for code tracing.

1.3.1 How it's designed

In order to scaffold writing, debugging, and explanation of meaningful and complex programs, purpose-first programming groups code into larger chunks called *programming plans* [138]. A programming plan connects a code pattern with a goal it achieves [134], drawing a direct connection between code and its purpose. In purpose-first programming, instruction will focus on plans in an authentic domain such as web scraping, rather than structures from a programming language.

1.3.2 Why it might work

The Structure Behavior Function framework tells us that programmers must have some conception of code's *behavior* (how code works) in order to create and understand novel programs. Typically, code tracing is the activity where novices learn about code behavior [140, 86]. However, my prior research showed that code tracing has a high cognitive load, and can even be seen as in conflict with certain learner self-beliefs. Some learners may be more motivated by a different way to understand code behavior, especially if it connects to the purpose for writing code, rather than the way a programming language works.

1.3.3 How I evaluated it and what I found

I developed a proof-of-concept curriculum that implemented the purpose-first programming approach in the domain of web scraping, and evaluated it with undergraduate non-major novice programmers who expressed a lower than average expectation of success or value for code tracing. Overall, I found that these novices were motivated to continue learning with purpose-first programming.

Participants were able to complete scaffolded code writing, debugging, and explanation activities in a new domain (web scraping with BeautifulSoup) after a half hour of instruction. An analysis of the participants' thinkalouds showed that they

used scaffolding about plan groupings and plan goals and subgoals to problem-solve. These learners showed evidence of thinking in terms of the patterns and goals that they learned with in the purpose-first curriculum.

I found that these novices felt successful during purpose-first programming because they could understand and complete tasks. Novices perceived a lower cognitive load on purpose-first programming activities than many other typical learning activities, because, in their view, plans helped them apply knowledge and focus only on the most relevant information.

Participants also expressed value for the purpose-first programming activities. They felt that what they were learning was applicable, and that the curriculum provided conceptual, high-level knowledge. For some participants, particularly conversational programmers who didn't plan to program in their careers, this information was sufficient for their needs. Other participants felt that purpose-first programming was a starting point, from which they could move forward to gain a deeper understanding of how code works.

1.4 Implications for programming learning

This work connects cognitive theories to theories of motivation in order to present a new approach to programming learning called purpose-first programming. My findings generate initial evidence for the positive impact of purpose-first programming on student motivation, for a population of students who lack motivation for an existing learning task: code tracing. The pedagogical approach of purpose-first programming creates a new pathway to programming learning for students not well-served by existing instructional norms.

The purpose-first programming approach is promising as a way to motivate learners who care more about what code can do for them than exactly how code works, such as conversational programmers [28, 157] and end-user programmers

[79, 37]. Future work should investigate how purpose-first programming can be effectively integrated into an instructional sequence in a classroom setting, which learner characteristics are correlated with a benefit from purpose-first programming, and what technology can support the development of purpose-first programming curricula.

CHAPTER II

Using the Structure Behavior Function Framework to Understand Programming Learning

For novices, understanding a computer program is a complex task. Code is made up of unusual symbols and words, which relate in intricate ways. Somehow, those characters instruct the computer to take action, but those actions are hidden deep inside the processor. Ultimately, the actions work together to achieve a goal – something the novice could explain in a natural language rather than a programming language.

In order to investigate *how* novices comprehend or design computer programs, we must know *what aspects* of computer programs there are to understand, and how those aspects relate. Knowledge of syntax and semantics is crucial for understanding code, but syntax and semantics alone cannot fully describe a program. Computer programs are designed with a purpose, to do work in the world. Every (real) program is a designed artifact, and knowledge about a program should include an understanding of the goals it achieves.

The Structure Behavior Function framework describes understanding of designed artifacts in three categories: knowledge of *structure* – what the artifact is made of, knowledge of *function* – why the artifact was built, and knowledge of *behavior* – how the artifact works to achieve its purpose. This framework has been

proposed and re-proposed by cognitive scientists [35], knowledge-based AI researchers [55], and design scientists [51]. It seems to express a foundational truth: that deep understanding of something that was designed involves knowing the *what*, the *how*, and the *why*.

In this thesis, I use the Structure Behavior Function framework as a common language to organize and relate several research areas in computing education. I also map popular theories of programming instruction onto the SBF framework, revealing their similarities, differences, and gaps in skill coverage.

2.1 The Structure Behavior Function framework

2.1.1 What is the Structure Behavior Function framework?

The Structure Behavior Function (SBF) framework is a general model for the understanding of a designed artifact. In the SBF framework, “knowing” and “understanding” an artifact involves more than simply understanding what the artifact is made of. For example, knowing that a circuit consists of wires that connect a light switch and a battery pack to a light bulb doesn’t fully capture understanding of that artifact. Someone who understands this circuit also knows that its *purpose* is to turn on a light easily, and that it *achieves* this goal by enabling electron flow through the filament of the bulb.

The SBF framework proposes that knowledge about a designed object exists in three distinct but interacting types: knowledge about the **structure** of the system, knowledge about the **behavior** of the system, and knowledge about the **function** of the system.

2.1.1.1 Structure is the parts of the artifact and their arrangement

The structure is what the artifact contains, or is made of. Examples of structures include the gravel in an aquarium [70], the clutch of a car [159], or the gas in a coolant system [55]. Structural elements are perceptually visible, and typically have important interconnections.

2.1.1.2 Function is the purpose of the artifact

It is the *reason* the artifact was created; *why* it does what it does. Functions describe high-level goals, for example, to keep bacteria count low [70], shift gears [159], or cool nitric acid [55].

2.1.1.3 Behavior is the mechanism by which the artifact works

It is how structures achieve their purpose. Many behaviors may need to occur in order to reach a single functional goal.

The mechanisms of behavior are often hidden from perceptual view. For example, the molecular interactions that cool an acid and the filtration of water by gravel occur on a microscopic level. The transfer of rotational energy from a clutch to an output shaft occurs deep inside the transmission of a car, and the forces at play may not be evident even if the components were visible.

2.1.2 How do structure, behavior, and function interact?

Designing an artifact involves assembling components and their interconnections in a way that fulfills a given purpose. In other words, *design is a translation from function to structure*. If the designer knows a design that meets their specifications, the design process is a simple lookup of a stored schema. When the design requires some inference, it becomes more complex.

The designer may use two types of knowledge to incorporate behavioral information into their design process. The designer may propose a set of expected behaviors that will achieve the function. This process of translating between behavioral and functional knowledge involves teleological inference [35]. The designer may also explore how certain components and their connections create mechanisms. This process of translating between behavioral and structural knowledge involves modeling knowledge.

Comprehension of a designed artifact could take on many forms, but in all cases, the structure is already provided, and the function and/or behavior is to be determined. This process also draws on teleological and modeling knowledge.

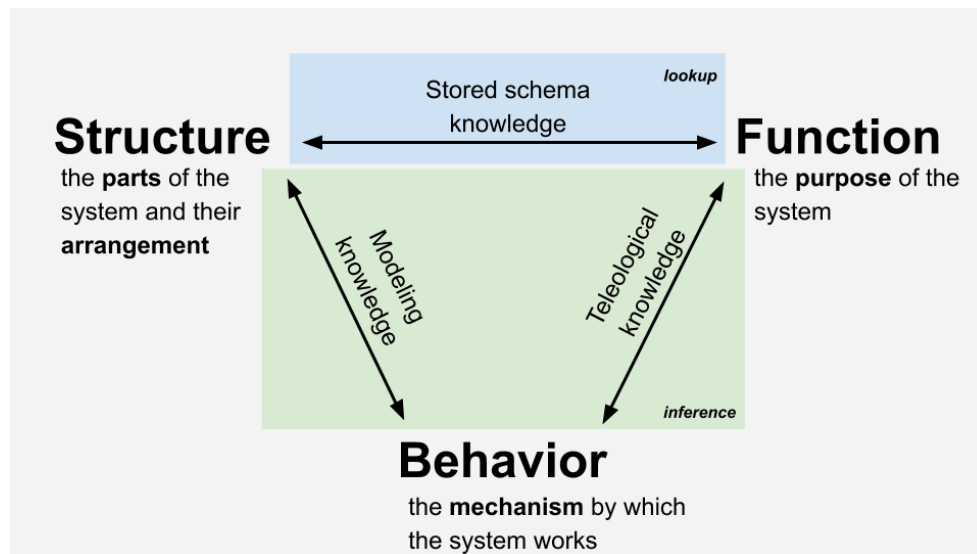


Figure 2.1: An illustration of the SBF framework

2.1.3 Where did SBF originate?

In *Sciences of the Artificial*, Herb Simon articulated seminal ideas about designed objects that would inspire the next generation of designers, cognitive scientists, and learning scientists [132]. Simon defined *artifacts* as objects that humans create in the service of their goals, and highlighted the duality of an artifact's function and the inner workings that achieve that function. As a result, Simon asserted that

the study of designed objects (any “science of the artificial”) must involve not only the study of how artifacts work and how they are structured (lines of inquiry that define the natural sciences), but also the study of artifacts’ intended goals.

2.1.3.1 Using SBF to model understanding of engineered devices

As cognitive scientists grappled with defining the knowledge necessary to understand artifacts like circuits and combustion engines, structure, behavior, and function were repeatedly identified as key concepts.

Weld proposed a general algorithm to produce descriptive summaries of complex engineered devices, as well as answers to questions about the devices [159]. The algorithm stored knowledge in four categories: knowledge about the device’s *role, function, structure, and mechanism*.

Weld’s categorizations organized knowledge in a way that allowed the algorithm to communicate many aspects of device knowledge. However, his categories were not always clearly defined. It is challenging to distinguish between “function” – what the device achieves *independent* of context (e.g. creating heat) and “role” – the goal of the device *within* a specific context (e.g. heating a room).

Later work clarified the distinctions between function, structure, and behavior. In a paper describing an algorithm for producing explanations of circuits, De Kleer defined function, structure, and behavior simply: “Structure is what the device is, and function is what the device is for, but behavior is what the device does” [35]. De Kleer describes a method for deriving function of a circuit from its structure, using behavior as an intermediate point. First, a qualitative description of behavior is inferred from circuit structure. Then, a process of teleological reasoning is used to infer the circuit’s function from the variety of behaviors (in a computationally expensive way).

2.1.3.2 Using SBF to model the design process

Other researchers moved beyond modeling understanding and began to articulate models for the design of a new artifact.

In his description of the “Function-Behavior-Structure ontology”, Gero claims direct inspiration from Simon [51]. He describes design as the process of producing a structure (a design specification) from a given function. Sometimes, the designer already knows a structure that will achieve the desired function. The interesting case for design occurs when the designer is not aware of such a mapping.

Gero asserts that the designer uses the function of the intended artifact to create a set of expected behaviors, and then a structure that creates actual behaviors. The actual behaviors are compared to expected behaviors, and discrepancies cause changes to the design. Gero considers much of a designer’s knowledge to be relational: in the form of connections between behavior and function, and connections between behavior and structure.

Goel and his colleagues used an SBF framework in a variety of knowledge-based AI tools that created new device designs from existing designs. KRITIK [54] and the related learning tool Interactive KRITIK [55] store information about engineered devices in structure, function, and behavior categories, and index the devices by their function. New device designs are created using case-based reasoning [81]. In the analogical reasoning system IDEAL, information about devices is also stored using the SBF framework [11].

The SBF modeling language Goel and colleagues created formalizes the SBF framework for use in a computational system [56]. In the SBF modeling language, each function points to a series of behaviors that achieves that function. A function is defined as the transition from one state to another state, while the behavior describes intermediate states between the initial and final states. This definition of function and behavior is hierarchical: a ‘function’ in one context may be part of a

behavior in another context. For example, the function of a water filter is to take unclean water and make it clean. From the perspective of the entire aquarium, a water filter creates behavior that contributes to the aquarium's function of keeping fish alive.

2.1.3.3 Similarities among SBF definitions

Understanding designed devices in terms of structure, function, and behavior has proven to be a powerful tool for many researchers. The act of design is viewed as a translation from function to structure, and the act of understanding is viewed as a translation from structure to function. Typically, behavior is viewed as *mediating* understanding and design. This isn't to say that structure and function can't be directly associated, but understanding or designing in this way is considered a trivial "lookup" [51, 35]. In the process of inferring function from structure or structure from function, behavior is a midpoint. When designing an artifact, the function is known, and the structure must be inferred. In order to infer the structure, the designer determines behavior that could create the desired function, and structure that creates that behavior. Similarly, by working out the behavior that a structure creates, and then the function that the behavior leads to, someone understands an artifact.

2.1.3.4 Differences between SBF definitions

Researchers vary in their definitions of structure, function and behavior. While some describe structure as sub-components and their arrangement [51, 35], others also consider the *interactions* between sub-components to be a part of an artifact's structure [159, 55].

Researchers define the behavior of artifacts in different ways. Most common is a focus on causal mechanisms [35, 55]. Weld sometimes describes behavior

in terms of constraints or metaphors [159]. In line with his roots in design science, Gero defines behavior in terms of an aggregate measure of characteristics like quality, time and cost [52], rather than the details of mechanics. A behavior of a software system, for example, is its response time.

2.1.4 What do we know about SBF in learning environments?

2.1.4.1 SBF can be a framework to assess learning

Cindy Hmelo-Silver and her colleagues have used the SBF framework to assess understanding of complex systems in a variety of contexts [69, 70, 72]. In these studies, interview data was analyzed by identifying mentions of relevant structures, functions, and behaviors. During the interviews, subjects were sometimes asked to draw a representation of the complex system, and then answer a variety of questions designed to elicit responses about structure, function, and behavior.

2.1.4.2 There are novice-expert differences in SBF understanding

Hmelo-Silver and colleagues were able to identify several trends in the facility of knowledge that novices and experts have about structure, behavior, and function.

Structural elements are most easily observed. In a comparison study between novices and expert aquarium hobbyists and biologists, structural elements of an aquarium (e.g. fish, gravel, plants) were identified at equal rates by experts and novices [70]. Novices were able to identify fewer functions or behaviors than structures, while experts identified more functions and behaviors than structures.

Hmelo-Silver and colleagues conclude that the order of understanding among the parts of the SBF framework is structure first, then function, then behavior last. This aligns with the perceptual availability of each component: while structure is straightforward for both novices and experts to observe, behavior is typically

hidden. Hmelo-Silver and colleagues also claim that knowledge of behaviors and functions is one of the “deep principles” which organize expert knowledge [70].

2.1.4.3 Using SBF to design learning activities

Building on the idea that experts often organize knowledge in terms of function, Liu and Hmelo-Silver used the SBF framework to design a hypermedia learning activity [92]. They compared learning outcomes after middle school students and pre-service teachers used a function-first curriculum or a structure-first curriculum about the human respiratory system. The two hypermedia learning activities had the same material, but the material was ordered and linked differently. In the structure-first curriculum, learners first saw the different parts of the respiratory system in a diagram, and could click on the structures to learn more about their behaviors and functions. In the function-first curriculum, information was organized by functional questions, like “How does oxygen get into the body?”. Within the units about the functional questions, related behaviors were linked, and the structures were linked from the behaviors.

While learning outcomes did not differ dramatically, there was evidence that the function-focused learners had a better understanding of “non-salient”, micro-level behavior, such as gas exchange and red blood cell activity. Middle school learners in the function-first condition also mentioned a greater number of behaviors in their post-tests.

SBF has also been used in the design of knowledge construction tools. Vattam et al. created a software tool where students model an aquarium system using structure, function, and behavior categories that create agent-based modeling simulations [154]. Later systems have expanded on this approach [76, 71].

2.2 What are Structure, Function, and Behavior in the context of programming?

People use computer programming to calculate, simulate, monitor, and create. Real programs are designed artifacts, coded with a goal in mind. The Structure Behavior Function framework applies, as it would to any other artifact.

2.2.1 Structure of a program

To identify the structure of a program, we need to ask questions like: *What “makes up” a program? What does a program “contain” ? and how are those sub-elements of a program arranged?* A program’s structure should be perceptually available, even to novices.

2.2.1.1 From a technical perspective

From the perspective of a parser, code consists of characters, arranged sequentially [4]. These characters are grouped into tokens, control structures, literals, and other elements prescribed by the language’s syntax rules. These elements can only exist in certain orders: syntax structures are defined by the production rules of the parser.

Textbooks, assessments, and research papers often take the view that syntax structures are the primary elements of programs, specifically, and programming knowledge, generally. For example, in a widely-cited survey study [83], Lahtinen and colleagues ask students and instructors to rate the level of difficulty of a variety of programming concepts. The list of concepts is mostly a listing of syntax structures, such as “arrays” , “parameters” , “pointers” , “loop structures” , and “selection structures” . The SCS1, a validated test of procedural programming knowledge, also uses syntax structures to organize its concepts [113], which were

gleaned from popular introductory programming textbooks [151].

2.2.1.2 From a human perspective

Kelly Rivers and her colleagues attempted to model student understanding of programming using syntax structures as core concepts, but were unable to do so [123]. Drawing on the Knowledge Learning Instruction (KLI) framework developed for use with intelligent tutoring systems [80]. Rivers et al. attempted to identify distinct skills (termed *knowledge components* in the KLI framework), using trace data from student code writing exercises. They considered the use of each syntax structure (e.g. for loop, function definition) to represent the application of a potential knowledge component. According to their framework, the rate of incorrect answers should decrease over time for a particular knowledge component, if the skill being measured is truly distinct from other skills. However, very few viable fits to expected learning curve models were found for the syntax structures. While the learning curve for a function definition decreased as expected, the learning curve for function calls and for loops did not. Using syntax characteristics alone to predict student learning came up short.

This result provides evidence that humans do not read code the way machines do. Humans do not read code sequentially, as a parser does [19]. Elements of code that are ignored by a parser or lexer are meaningful to humans, such as white space [119], variable names [50], and comments [119]. We know that humans perceive chunks of code that include many syntax elements as meaningful subcomponents of their programs [136].

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 [138].

Soloway and Ehrlich [136] showed evidence for plan schemas in a series of experiments, influencing the fields of program comprehension and programming education. They compared novice and expert programmers' performance on fill-in-the-blank problems and recall problems for "plan-like" code and "un-plan-like" programs, which had some differences from a similar plan-like program. Both experts and novices performed better on plan-like problems than un-plan-like problems. Experts performed better than novices on plan-like problems, but on un-plan-like programs, the difference between experts and novices was not significant. Also, the code that participants used to fill in the blanks for un-plan-like problems often matched what would have been expected for a corresponding plan-like program.

Rist also explored plan schemas, finding further evidence of plan knowledge in both novices and experts [122]. Experts used only plans to organize their understanding of programs, while novices resorted to description of syntactic structures when code became more complex [122]. Rist [120] also tracked the development of plan schemas in novice programmers over time, observing the frequency of "top-down" design, where the goal is primary and the associated plan is already known, versus "bottom-up" design, where the plan is constructed on the fly. He found that novices started with a bottom-up approach, identifying some key parts of a plan, but refining it over time. After they successfully completed a plan, its schemas was available for later retrieval. Rist noted that schemas can exist at different levels of abstraction, and proposed four levels of detail: a line of code, a simple plan, a complex plan, and a whole program.

The concept of programming plans has had a re-emergence in computing education research. The foundational work in programming plans most often included a procedural programming perspective. However, the characteristics of the functional paradigm provide a different perspective on programming plans. In a functional language, sub-goals of a programming task can often be relegated to various

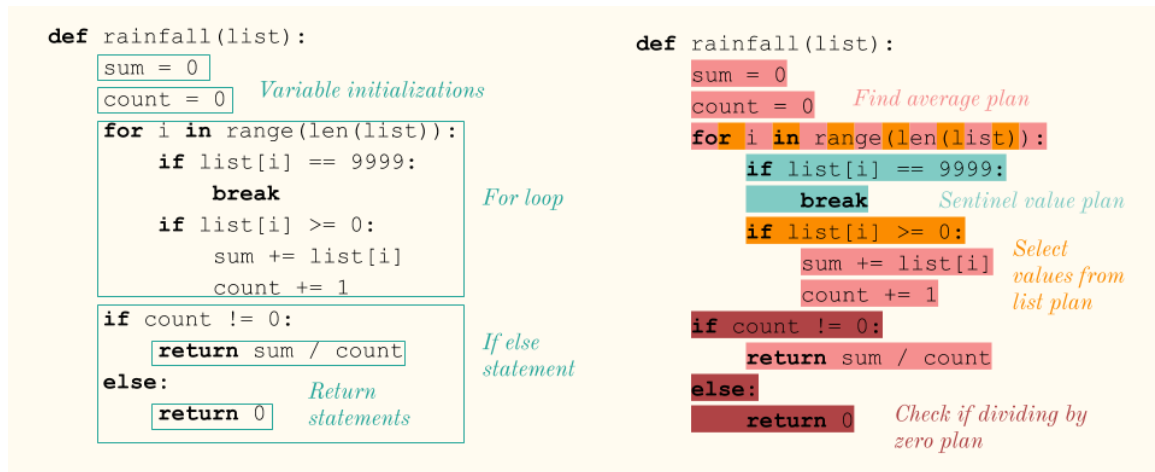


Figure 2.2: The rainfall problem with syntactical substructures highlighted (left) and plan substructures highlighted (right).

functions, which can then be easily composed [47]. This lessens the challenge of composing a variety of programming plans into a final, more complex program.

Programming plans have also received recent attention in the context of code complexity. Duran et al. expanded upon Soloway’s concept of the plan tree, incorporating syntactic elements as the leaf nodes of a plan tree [39]. Called the Cognitive Complexity of Computer Programs framework, this approach formalizes two metrics for code complexity: plan depth (levels of the plan tree) and maximal plan interactivity (the number of sub-plans that overlap during execution of a parent plan).

While the concept of programming plans has psychological validity, there are challenges to fully understanding how plans represent code structures. While there are a limited number of syntactic structures defined for each programming language, it would take significant effort to define enough programming plans to cover code typically written by novice programmers. While it’s easy to detect syntactic structures (the compiler is doing it anyway), it is unclear how to automatically detect programming plans. Also, the process of combining plans to create larger programs is not well-understood in novice programmers, beyond the fact

that it is the source of many errors [145].

2.2.1.3 Conclusion

Structure certainly includes code syntax, but syntax can't explain all aspects of code structure. As a programmer's knowledge increases, larger and more complex chunks, such as code plans, should be considered as structural units.

2.2.2 Behavior of a program

Code structures instruct a computer to achieve some function. According to the definition of the SBF framework, behavior consists of the *mechanisms* that translate structure into function. Behavior is *how* an artifact works.

2.2.2.1 From a technical perspective

In order to execute a program, a computer undertakes many, many steps. Consider the standard process to execute a C program [4].

1. A lexer breaks the code into a stream of tokens
2. A parser determines whether those tokens adhere to the rules of the programming language
3. A compiler translates the code into a series of low-level machine instructions
4. The central processing unit executes the machine instructions, using registers and the arithmetic logic unit.

This behavior is highly complex and involves many distinct systems. In addition, the process can vary due to the programming language (e.g. when a language is translated into bytecode rather than machine instructions, as in Python) or, to a

lesser extent, due to the details of the machine on which the code is executed. Little to none of the code execution behavior described above is self-evident from the syntax of code. Like behavior in other systems, program behavior is hidden and perceptually unavailable.

2.2.2.2 From a human perspective

It is possible to have a useful understanding of how code executes without understanding the details described above. Computing educators depend on abstractions of the more complex and lower-level processes as they teach introductory programming. They leave out details about tokenization, parsing, memory, and execution in order to narrow the material that students must learn. The way that an instructor describes the process of code execution is called a *notional machine*.

As early as the mid-1970s, researchers noted that by providing a concrete model of the way code was executed, some learner outcomes improved [101]. Learners who “simulated code execution” before learning language structures performed better on far-transfer tasks, but worse on near-transfer tasks.

Later, theory about modeling code execution was further developed. Along with Tom O’Shea and John Monk, Du Boulay coined the term “the notional machine”, defining it as “an idealized, conceptual computer whose properties are implied by the constructs of the language” [15]. They also articulated the idea that programming language designers can choose the level of abstraction that understanding their language requires. They suggested that programming languages with a “small number of parts” were ideal for novices.

Juha Sorva reviewed prior work related to notional machines in his dissertation [139]. He noted that there are certainly different notional machines for different programming paradigms, which differ in whether variable states change or are immutable. However, notional machines can also differ for different programming

languages within the same paradigm. There are even different notional machines for a single language, depending on the level of abstraction. When understanding code using objects, it makes sense to think about code behavior differently than when understanding code that does sequential calculations.

While there are many possible notional machines, in practice, the way that program execution is illustrated for novices is remarkably consistent [141]. For the dominant procedural and object-oriented paradigms, variables and their changes are shown, typically in boxes. As each line of code is executed, the variable boxes are updated appropriately.

Most research about notional machines has taken place in the procedural programming paradigm. Notional machines for functional languages take a different shape, and have been articulated as a substitution model [48]. A functional orientation allows for programming plans to be represented in a single function, and easily combined.

Computing education researchers who study the notional machine frequently underscore its importance. DuBoulay expressed a view that learning programming is primarily understanding the notional machine [38]. Sorva argued that knowledge of the notional machine meets criteria to qualify as a “threshold concept” [140]. A threshold concept changes a learner’s understanding of concepts learned afterwards [104]. The great many number of misconceptions that computing education researchers have found about the notional machine supports its importance. Sorva lists 162 misconceptions about procedural notional machines in his dissertation [139].

While the most common description of code behavior is having a mental model of the notional machine, other researchers sometimes call this knowledge of a programming language’s semantics¹. For example, when describing the tool PLTutor

¹Confusingly, the term semantic knowledge is also sometimes used to describe code patterns that are associated with a particular purpose (e.g. [101]).

and the way that it simulates the notional machine, Nelson, Xie, and Ko use the phrase “*execute semantics*” rather than “*execute the notional machine*” [110]. Sorva claims that multiple notional machines may be involved in understanding a language, but the term semantics is typically used as if there is only one way to understand semantics. In PLTutor, a high level of fidelity to the actual process of the Python interpreter was valued, and as a result, the method of illustrating code execution was relatively low-level.

2.2.2.3 Conclusion

In programming, behavior of code is the action of the notional machine on that code. The level of abstraction of the notional machine may vary in different settings and with different programmers.

2.2.3 Function of a program

The function of a program is the program’s *purpose*, the *goal* of the program, and the *reason* the program was designed.

2.2.3.1 From a technical perspective

The computational machinery that executes programs has no knowledge of why a program was created. A program’s function is always bound to the human side of program development. Software engineers have created many mechanisms to map from the human to the technical. Design specifications, documentation, and test suites that serve as “executable specifications” all describe the goals of programs. These approaches often focus on the inputs and outputs of programs in order to create a more formal description of code functionality.

2.2.3.2 From a human perspective

Begel and Simon found that new software developers often bemoaned the lack of documentation on their projects, seeking more assistance in understanding code [9]. Somewhat more experienced developers realized that documentation has limitations, and instead searched for people to talk to.

Code function defined in terms of initial state and final state is consistent with the way Goel's SBF modeling language expresses function [56]. However, from the perspective of a programmer, the format of the parameters and return values are also design choices. Before specific parameters are chosen, a higher-level, more abstract, more plain-spoken goal was likely articulated by a programmer or designer.

Computing education researchers have studied novice programmers' ability to "explain" code "in plain English", finding that programming learners explain code in a variety of ways. Members of the BRACElet project [30] adapted the SOLO taxonomy [12] to categorize these responses [91]. In this framework, a "unistructural" response to code explanation question describes only a subset of the code; a "multistructural" response describes most of the code, but with a focus on how code executes; and a "relational" response provides a summary of the code's purpose. Lister and his colleagues found that novices rarely responded with a relational response when asked to describe new pieces of code [91]. Experts, on the other hand, typically used relational responses.

2.2.3.3 Conclusion

The function of a program is best described as a human language description of the program's purpose. Some programs do not have a function, such as programs designed only to illustrate code behavior.

2.2.4 Connections between structure, function, and behavior.

According to the SBF framework, the ways that a human *translates* between structure, behavior, and function are key to understanding expertise in the creation and comprehension of a designed artifact. What actions in programming include the translation of structure to behavior, behavior to function, and structure to function?

2.2.4.1 Tracing code: structure to behavior

According to Sorva, “Tracing a program requires the programmer to keep track of the state of program execution, that is, to simulate the job of the notional machine.” [140]. Nelson, Xie, and Ko agree: “the knowledge needed to learn program tracing is not the abstract formal semantics for a language, but the semantics as actually implemented in a language’s interpreter, mapped to a notional machine to facilitate comprehension.” [110]. Tracing connects code *structure* to code *behavior*.

Teaching tracing skills explicitly seems to have a positive effect on novices’ ability to simulate code execution. Sorva found positive effects due to use of program visualization tools in his review of such systems, although the number of controlled evaluations was limited [141]. Use of PLTutor, a tool that provided direct instruction about a low-level notional machine [110], improved student outcomes on the SCS1 [113]. Demonstration of a code tracing method was associated with improved scores on a programming exam as well as on selected SCS1 problems in a small study [164].

2.2.4.2 Explaining code: structure to function

Code explanation, typically evaluated through “explain in plain English” questions, requires students to look at code structure and then find code function. It involves a translation between code *structure* and code *function*. There appears to

be a trend towards higher levels of abstraction for more proficient programmers. Sudol-DeLyser [147] found that higher proficiency students were more likely to use abstract statements during think-alouds while writing code. She also found that higher proficiency students made more transitions between levels of abstractions.

Analyses of program comprehension have identified *both* tracing and chunking as methods that experts use to understand programs [23]. Novices also use a variety of tracing or pattern recognition approaches [49]. It is unclear whether one approach is superior to the other for novices; studies have found significant error rates for both approaches [49, 34]. While the ability to accurately recognize patterns may be a sign of expertise, incorrect pattern recognition is characteristic of many novices.

The BRACElet project is known for their investigations of a wider variety of programming skills than code writing alone [30]. They studied code tracing, code explanation, and code writing skills, and have consistently found correlations between these three skills. A stepwise regression analysis suggested a hierarchy of skills, where code tracing and code explanation could predict code writing [93]. However, no clear causal link between any of these skills has been identified.

2.2.4.3 Writing code: function to structure

Writing code requires translating a human language description of an objective and creating code that achieves that objective. Writing code translates code *function* to code *structure*. Programming plans suggest one method that programmers make this transition from function to structure. Each programming plan is associated with both a function (the plan goal) and a structure (the plan code) [134].

2.3 Three theories of programming instruction through an SBF lens

While writing programs is the prototypical activity of a programming learner, many other coding skills can be taught and assessed. *Theories of programming instruction* answer the questions *What are the important skills programming learners should understand?* and *How do these skills relate to each other?* Theories of programming instruction delineate a number of programming skills, and sometimes also a suggested ordering or hierarchy of those skills. They provide a guideline for what should be taught and when.

The SBF framework allows us to compare and contrast theories of programming instruction, by mapping the skills the theories describe onto relationships between structure, behavior, and function.

2.3.1 The “Neo-Piagetian” hierarchy of programming skills

2.3.1.1 Description

Inspired by neo-Piagetian stages of cognitive development, Raymond Lister proposed several sequential stages to describe novice programmers’ level of understanding of programming topics [86]. For each “concept” in an introductory programming course, the learner is categorized into a stage of the model. The categorization is not strict, however: while one stage is typically dominant at a particular time, the stages may overlap.

The stages are listed here in order of increasing development. These definitions synthesize the definitions from the initial description of the theory [86], a later refined description of the theory [87] and Donna Teague’s dissertation [149].

- In the **Sensiomotor stage, aka Pre-tracing stage**, a learner understands 50% or less of program execution rules for the topic. They are unable to reason

“in plain English” about what code does.

- In the **Preoperational stage, aka Tracing stage**, a learner understands more than 50% of program execution rules for the topic. In order to understand “what code does”, the programmer traces the execution of code on a concrete example and then makes an inference based on the change between the input and output.
- In the **Concrete operational stage, aka Abstract tracing stage**, the learner can reason about some parts of code without tracing. They can write code that involves a small change to an existing example, such as reversing an operator.
- In the **Formal operational stage**, the learner has reached the level of an expert on that topic. They reason about code by reading code, not by tracing it. They can hypothesize about code well enough to debug.

Lister uses ability in code explanation and code tracing as key indicators of programming understanding. The major finding of the BRACElet project – that code tracing skill correlates with code explaining skill and code writing skill – is consistent with this hierarchy. A later microgenetic study that tracked the development of a single student provided some additional support [150].

2.3.1.2 Through an SBF lens

In Lister’s model, a programmer must understand the details of how code works before being able to state the purpose of that code. In SBF terms, this means knowledge that translates structure to behavior (tracing) is a prerequisite to knowledge that translates structure to function (explaining code). Knowledge that translates behavior to function is not explicitly discussed in this model.

Lister describes his stages as occurring for “each concept”. A key question is “what are those concepts?” Lister and his co-authors explicitly state that a new concept is a new syntactic structure: “...a student progresses from sensorimotor, to preoperational to concrete operational *when the programming constructs to which the novice is exposed do not change* [150]. For example, when the student under micro-genetic study learned about arrays, Teague and Lister found that he lacked tracing ability with arrays and regressed to an earlier stage. From an SBF perspective, the concepts come from a particular type of code structure, the syntax.

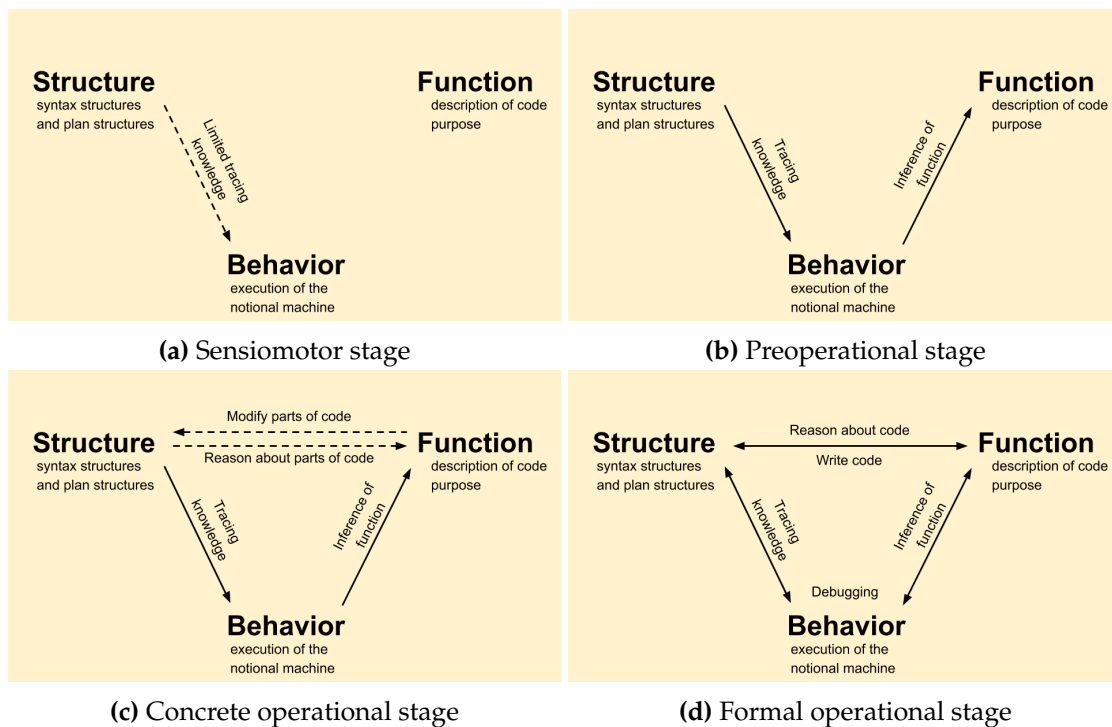


Figure 2.3: The abilities of learners in Lister’s Neo-piagetian stages, mapped to the SBF model

2.3.2 Xie et al.’s theory of programming instruction

2.3.2.1 Description

Xie et al. [163] identify four distinct skill areas within programming, and suggest that they should be taught in a particular order. Inspired by the findings of the

BRACElet project and of the various cognitive scientists and computing education researchers who studied programming plans, they claim that code reading should always precede code writing, and that understanding the notional machine should always precede knowing programming plans.

The four skill areas and their order are:

1. **Reading semantics:** Given code, trace the action of the notional machine
2. **Writing semantics:** Given a detailed description of notional machine action, write correct syntax
3. **Reading templates:** Given code, identify its programming plan and associated objective
4. **Writing templates:** Given an objective, use appropriate programming plans to write code

Xie et al.'s hierarchy is unusual in its incorporation of human language description of notional machine action. Learners are expected to be able to describe notional machine action at the reading semantics stage, and be able to read and understand a description of notional machine semantics in the write semantics stage (Figure 2.4, S1 and S2). When writing code in the write templates stage, learners are expected to translate an objective into this human language description of notional machine action, and then into code (Figure 2.4, S4).

As a result, Xie et al. believe that weak understanding of how code executes may result in learners “not recognizing a template in the code, recognizing a template but not recognizing it was incorrect, or recognizing the wrong template.” [163]. From this perspective, plan recognition is not solely pattern recognition, but also involves mental execution of the notional machine.

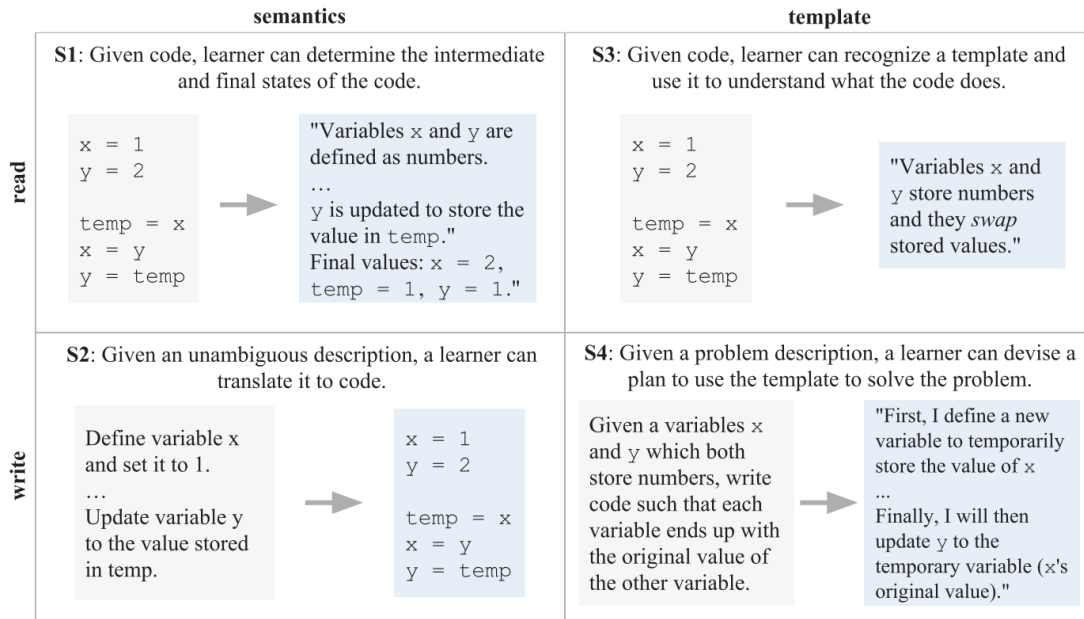


Figure 2.4: Xie et al.'s four skills from their theory of instruction [163]. ©Benjamin Xie

For what unit of a curriculum do the four stages need to be followed? Should the notional machine action for all syntax structures in a language be learned before moving on to any programming plans? Although this issue is not addressed explicitly, in the sample curriculum that Xie et al. implement and evaluate, they cycle through the four skills when they teach a new group of syntactic structures, such as relational operators or conditional statements (pg. 14). The plans taught in each cycle only use syntactic structures explicitly covered in a prior tracing lesson.

Since inspiration for both Xie's theory and Lister's theory comes from the findings of the BRACElet project, it's not surprising that both skill hierarchies put code tracing before code explanation. Lister doesn't discuss how his stages relate to writing skill, except that he believes novices who haven't reached the formal operational stage cannot write code effectively [86]. Xie's hierarchy addresses code writing skills, which is possible because of the incorporation of programming plans into the theory.

2.3.2.2 With an SBF lens

We can restate this hierarchy of skills in terms of the SBF framework:

1. **Reading semantics:** translate from structure to behavior
2. **Writing semantics:** translate from behavior to structure
3. **Reading templates:** translate from structure to behavior to function
4. **Writing templates:** translate from function to behavior to structure

This theory of instruction prioritizes knowledge about behavior over knowledge about function. It also claims that knowledge about function cannot be fully understood without knowledge about behavior.

From what we know about structure, function, and behavior knowledge in novices, does it make sense to put so much early emphasis on behavior? We know that knowledge about behavior has been the most challenging for novices to learn in other contexts [70]. On the other hand, experts distinguish themselves from novices by having a greater amount of knowledge about behavior [70]. Xie et al. believe behavior should be taught explicitly and early. If novices are able to incorporate behavior knowledge early on, this could be an efficient path towards expertise.

The writing semantics skill covers the translation from behavior to structure, which is rare in programming education. Identifying execution dynamics for *provided* code is much more common. Xie et al. focus on building this skill so that it can be used in the later writing templates skill.

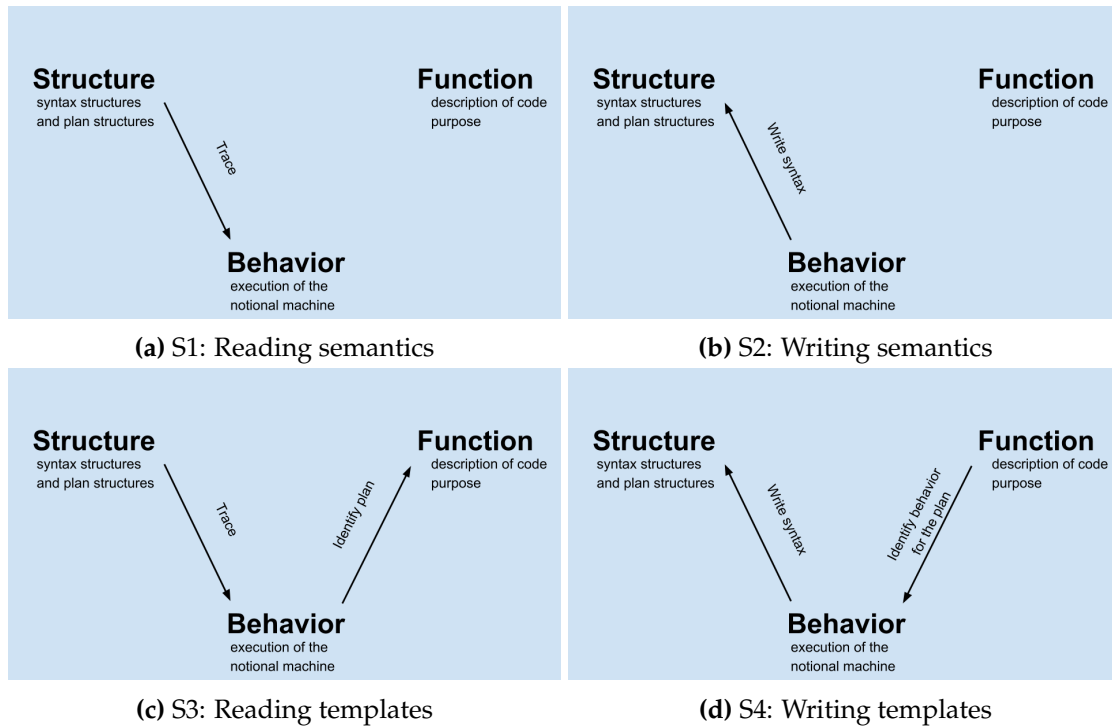


Figure 2.5: Xie et al.'s instructional stages, mapped to the SBF framework

2.3.3 Schulte's Block Model

2.3.3.1 Description

The Block Model [129] describes a matrix of skills related to code comprehension. Two dimensions create twelve “blocks” of knowledge about code. One dimension is the level of *abstraction*: “macro structure” , at the level of the entire program; “relations” , between sub-parts of the program; “blocks” , meaningful sub-parts of the program; and “atoms” , elements like tokens and syntax structures, defined by the programming language. The other dimension is the *aspect* of the program: “text surface” , “program execution” , and “functions, goals of the program” .

Schulte notes that the ultimate goal of code reading is to get from the lower left corner of Table 2.1 (language elements) to the upper right corner (understanding the goal/purpose of the program). Schulte believes that learners can connect skills

	<i>Text surface</i>	<i>Program execution</i>	<i>Functions</i>
<i>Macro structure</i>	Overall structure of the program text	The “algorithm” of the program	The goal/purpose of the program
<i>Relations</i>	References between blocks, e.g.: method calls, object creation, accessing data	The sequence of method calls	How subgoals are related to goals, how function is achieved by subfunctions
<i>Blocks</i>	Regions of Interest (ROIs) that syntactically or semantically build a unit	Operation of a block, a method, or an ROI	Function of a block
<i>Atoms</i>	Language elements	Operation of a statement	N/A - goal only understandable in context

Table 2.1: The Block Model, adapted from [20].

in adjacent blocks in order to transition between the different types of knowledge. The path to move between language elements and code purpose involves horizontal movement between aspects of the program, as well as vertical between levels of abstraction. There are several paths to get from reading code to understanding its purpose, but all paths involve an increasing level of abstraction and a transition through code behavior.

There is no prescription for the order in which different blocks should be addressed in the classroom. This is by design. Schulte states that since learners naturally build abstract schemas on their own, and are working from an individualized set of prior knowledge, “teaching should focus on supporting [the abstraction] process, instead of focusing on providing all necessary steps in a predefined order, with predefined bits of information.” (pg 150)

Schulte also sees the Block Model as supporting different “types” of understanding at different levels of abstraction and about different aspects of code. He notes that understanding code from different perspectives shouldn’t be considered a misconception, but instead a jumping-off point for connecting to other blocks in

the model.

An initial study with pre-service computing teachers found that participants who learned about the Block model used language from the model while planning a lesson [129]. However, there were not major differences in lesson quality between preservice teachers that learned about the model and those that did not. Both groups did discuss the duality between code structure and function while planning.

This model distinguishes itself from other models by identifying multiple levels within code text, code execution, and code purpose. Consistent with the understanding that a notional machine can be described at multiple levels of abstraction [140], Schulte describes how the purpose of code and the text of code can be understood at multiple levels of analysis as well.

Schulte consciously limits the scope of the block model in order to maintain a reasonable level of detail without over-prescribing activities that may stifle instructors' ability to individualize instruction. This is a well-considered choice, but it does mean that the block model is limited in a couple ways. Schulte only considers code reading in the model, and it's unclear if the model is applicable in any way to code writing. Also, there isn't a hierarchy of skills, or a suggested order. This is by design, but does mean that instructors cannot use the model "off-the-shelf" .

2.3.3.2 With an SBF lens

Although the SBF framework is not cited as inspiration, the Block Model is remarkably consistent with the SBF framework. Text surface is the structure, program execution is the behavior, and the purpose or goal is the function. Schulte also considers code behavior to be the crucial but hidden and complex moderator between code structure and code function.

Schulte's key insight is that structure, function, and behavior can exist on many

levels. This hierarchical interpretation is consistent with the approach to function taken by the SBF modeling language, where the function of one component can be considered part of the behavior of another component. Programming inherently involves many levels of abstraction, so this use of multiple levels is appropriate. This use of multiple levels provides more rhetorical power than Lister's neo-Piagetian stages, or Xie et al.'s theory of programming instruction.

2.3.4 Discussion

Mapping these theories onto the SBF framework exposes many similarities. All three theories of instruction describe how knowledge of code behavior is important for connecting code structure to code function. In Lister's Neopiagetian stages, learners develop tracing knowledge before the ability to explain or write code. In Xie et al.'s theory of instruction, the process of code execution should be taught first, as it is believed to be a crucial prerequisite for explaining or writing code. In the Block Model, code comprehension requires understanding of program execution.

The Neopiagetian stage model and Xie et al.'s theory of instruction both propose a progression where code function is only incorporated in later stages. Connecting structure knowledge to behavior knowledge is first; connecting structure knowledge to function knowledge comes later.

All three theories have another commonality: there is no discussion of how motivation may interact with learning. All three theories have been developed and tested in contexts with computer science majors or students who opted in to a computer science course about introductory programming. These learners already decided that programming knowledge is important for its own sake.

Approaches like contextualized programming education, which was primarily developed for non-majors, emphasize code function much more strongly. In con-

textualized computing, programming learning is motivated by code's application to real-world problems. Media Computation [62], a course where learning programming is motivated by applications like photo editing and sound manipulation, is one of the few consistently successful curricular innovations in computing education [116]. Media computation was designed for liberal arts students who were required to take introductory computing, but did not see much use for programming for their chosen careers. Compared to a traditional introductory computing course, students who took Media Computation expressed a greater sense of the relevance of computing for their future, and stayed enrolled in the course at higher rates [63].

It's unclear whether students in Media Computation learned as many programming skills as students in traditional introductory computing courses [63]. However, even if an approach like Xie et al.'s is more efficient in teaching programming skills, it doesn't matter if unmotivated students don't stick around when they can't see the applicability of the course content to their future careers.

2.4 Conclusion

This thesis applied the Structure Behavior Function framework to the understanding and design of computer programs. The meaning of structure, behavior, and function was determined in the context of programming, and the ways that a program's structure, behavior, and function relate were mapped onto programming skills. By applying what we know about novices' program comprehension and program design onto this framework, I have related several threads of computing education research, such as the study of notional machines, programming plan theory, and the natural language explanation of code. By describing theories of instruction in terms of the framework, I have determined similarities and differences between the ways that we teach and the ways that we expect learners to

understand material.

More broadly, the Structure Behavior Function framework provides a common vocabulary for discussion about programming skills, assessment, and philosophies about how to teach best. It allows us to compare and contrast theories and perspectives, and propose new research directions. Computing education researchers work with humans who want to achieve goals in the world using their programs. The Structure Behavior Function framework gives our community a language to not only talk about how programs work and what they are made of, but also why they were written, and what they achieve.

CHAPTER III

Novice Rationales for Sketching and Tracing, and How They Try to Avoid It

Prior research has shown that sketching out a code trace on paper is correlated with higher scores on code reading problems. Why do students sometimes choose not to draw out a code trace, or if they do, choose a different sketching technique than their instructor has demonstrated? In this study, we interviewed 13 CS1 students retrospectively about their decisions to sketch and draw on a recent programming exam. When students do sketch, we find that their sketching choices do not always align with a strict execution of the typical notional machine for CS1. Sketching choices are 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. When novices don't sketch, they often report that they've identified the goal that the code achieves. In either case, novices are searching for the functionality of code, rather than merely tracing its behavior.

3.1 Introduction

When novice programmers solve programming problems, they sometimes use pen and paper. They might draw out variable states, re-write code snippets, or

work through calculations. Called sketching [33], annotations [102], or doodles [88], these drawings are an external representation that many students find useful. Research supports the utility of sketching. Sketching is correlated with greater success on code reading problems, with sketched traces being the most successful [88, 33].

Prior research on student sketching has involved qualitative and quantitative analysis of scratch sheets. However, the motivations of the sketchers remained a mystery. If sketching is associated with success, why do students only occasionally use it? Why do students sometimes stop part-way through a code trace, behavior that has been associated with particularly low success [33]? Student sketches differ in several ways from the more formal memory diagrams proposed by instructors [74], or the graphics in program visualization tools [141]. We have shown that students' sketches often have a different style than the ones used by their teachers [33]. Students' motivation for why and how they sketch is unclear.

If we knew more about the reasons students sketch and the approaches they favor, perhaps we could design sketching or other visualization techniques that students are more likely to understand, use, interact with, and learn from. In this study, we perform interviews with CS1 students to explore the following:

- Why do novice programmers choose to sketch or not sketch on different problems?
- What rationales do novice programmers provide for their sketching styles?
- Why do novice programmers choose a different sketching technique than their instructor?

3.2 Background

3.2.1 Tracing and other ways novices read code

Tracing a piece of code involves tracking memory states over time as the execution of code is simulated. Tracers are running a mental model of the notional machine [140], acting as “human compilers”.

We want students to trace code because it can lead to students’ greater accuracy during problem-solving [164], and it may play a role in improving learning about the notional machine, a concept where students have many misconceptions [139]. But students do not necessarily trace because it is good for them, and they may take other approaches to solve problems. In this paper, we ask students why they trace code and why they trace the *way* that they do.

We do know that students voluntarily trace when faced with complex problems. The Leeds Working Group [88] asked students to read and predict the execution of programs that used loops and arrays. Students’ scratch paper was collected and analyzed, finding that many students had sketched a code trace.

Fitzgerald et al. [49] performed cognitive walkthroughs with introductory CS students as they solved problems with the same problems used in the Leeds Working Group exam. Using grounded theory, the researchers identified 19 strategies students used while solving code reading problems, including close tracing (the most frequent) as well as various types of pattern recognition (e.g., recognizing that this program was like one seen before). However, the use of a certain strategy did not predict success.

Fitzgerald et al.’s interviews have also been analyzed with the SOLO taxonomy as a theoretical lens. Lister et al. found that students were more likely to describe using a multistructural approach, where they discussed the function of multiple pieces of code, rather than a relational approach, where they discussed the code as

a whole [91]. Other literature has reported that novices focus on the pieces rather than the program purpose. For example, expert programmers are more likely to use abstract mental representations that focus on the goal the code achieves, while novices are more likely to focus on details of how the code executes [3].

Vainio and Sajaniemi identified factors that they felt inhibited novice code tracing [153]. Tracing requires attention to detail, which can lead to cognitive overload and mistakes (e.g., tracking and using variables) and poor use of external representations. Like Fitzgerald et al.'s pattern recognition, Vainio and Sajaniemi found that students assume functionality from a syntactic structure that matches a previously seen pattern.

3.2.2 When and why do students sketch?

We know that while solving code reading problems, sketching a code trace correlates with success [88, 33]. However, students are more likely to create these types of sketches on some types of code reading problems than others [102]. For questions that are not about code reading, like Parsons problems [114], students rarely sketch anything [33]. Also, students' sketches often does not use the types of diagrams that instructors teach [160, 33].

Analysis of the Leeds Working Group data showed that students' sketching varied widely from institution to institution, from as as low as 28% of problems sketched to as high as 92% [102]. Students consistently sketched more often on code prediction problems than on code completion problems.

Sketches have the potential to not only offload cognition, but also to better coordinate thought [78]. Certainly, sketching out variable states provides a persistent way to track information too complex to remember accurately. But also, in the unconstrained space of a sheet of paper a sketcher can re-arrange information to bring key referents closer together, or make certain information more explicit than

it was in a prior form [78].

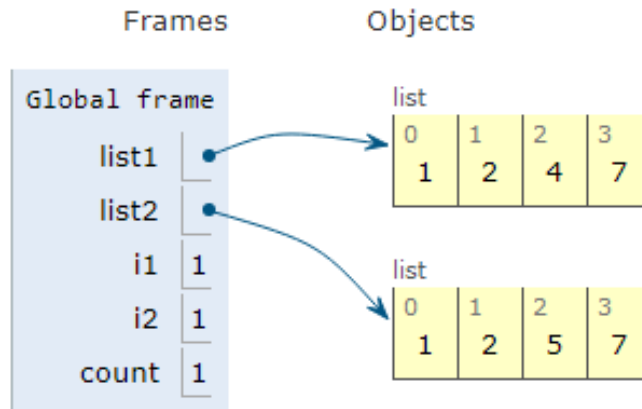
3.2.3 What external representations are common in novice programming?

The process of code execution, known as the action of the notional machine [38], is not self-evident from code syntax or program output. Systems that teach *code tracing* attempt to make this challenging, hidden process visible [110, 164, 67].

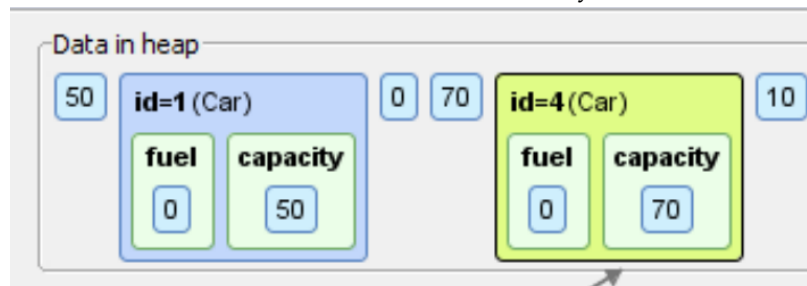
Software that displays program visualizations is a common approach in the effort to help students form accurate mental models of program execution. Systems like the Online Python Tutor [60] and UUhistle [142] demonstrate the action of the notional machine that a student can step through line by line (see Figure 3.1a,b). The visualization style of the great majority of these systems is remarkably consistent in nearly all such systems [141]: variable values are encased in boxes, and prior values are erased and overwritten.

A notable exception is PlanAni [124], a visualization tool that illustrates variables based on their “roles” [125], or functionality. The “most-recent-holder” role and the “stepper” roles are both illustrated with previous values (see Figure 3.1c).

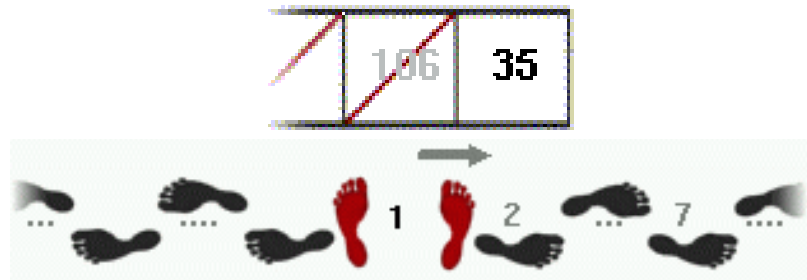
Empirical studies of visualization systems for use in introductory programming and in algorithms have had inconsistent results [141, 75]. Visualizations do not always help students, possibly because watching a visualization is a passive learning experience. An *interactive* process for visual program simulation is more likely to be effective [139]. This may involve an interactive GUI interface, like that of UUhistle [142], or potentially a sketching interface. To that end, viewing visualizations that students create for themselves may suggest new directions for the design of these tools.



(a) Variables and lists illustrated in Online Python Tutor [60]



(b) Variables and instances illustrated in UUhistle [142]



(c) A “most-recent-holder” and a “stepper” variable illustrated in PlanAni [124].

Figure 3.1: Variable visualizations in program animation tools

3.3 Method

CS1 students from a large research university in North America participated in a computer-mediated experiment, using their own laptops in a supervised setting. The experiment took place during the 10th week of a 16 week semester.

The analysis presented here is from five multiple-choice code reading questions in the pre-test portion of this experiment. These questions were written in Python, and tested CS1 knowledge on lists, loops, and conditionals. In these questions,

Table 3.1: Code plans appearing in code reading questions.

#	Apparent code goal	Achieves goal?
1	Find minimum in a list subset	Yes
2	Find number of common list elements	No
3	Count appearances of target in a list subset	No
4	None (arbitrary arithmetic)	N/A
5	Find average of list subset	Yes

participants were asked to determine the result after the code had executed. Some questions were examples of common programming plans (see Table 3.1), although the goal of the code was never specified. Students had 15 minutes to answer the five questions.

During the test, participants were instructed to use provided pens and blank scratch paper (labeled with their unique identification number) if they wished to draw. Participants were instructed to return their scratch paper to the experiment administrators after completion of the test.

3.3.1 Class observations

All participants had the same CS1 course instructor. The first author attended two class sessions during weeks 5 and 6, when lists were introduced. Field notes about instructional strategy were taken, and the sketches created by the instructor during these sessions were collected.

3.3.2 Interviews

Twenty-six of 167 test-takers consented to be contacted about a follow-up interview. A subset of those who consented were contacted, in order to maximize the variability the interviewees scores on the pre-test. Twenty-one students were contacted, and 13 agreed to participate in an interview. Of those 13, three got four questions right, six got three questions right, two got two questions right, one got

one question right, and one got no questions right.

Interviews were semi-structured, lasted 30 minutes, and occurred in weeks 14, 15, and 16 of the semester.

During the interviews, interviewees' scratch paper from their pre-test was presented, along with the problems from the original exam. Interviewees were given time to re-familiarize themselves with their work, and then asked to describe their choices to sketch or not sketch on each problem. Then, the instructor's sketches were presented, and the students were asked to describe why they chose to sketch similarly or differently than their instructor.

Interviews were recorded and transcribed. They were coded thematically by two coders (the first and third authors), working individually.

3.4 Why sketch (or not)?

3.4.1 Goal and pattern recognition

3.4.1.1 Identifying a goal

Most questions matched or appeared to match familiar coding plans (see Table 3.1). For those problems, several students claimed to have identified the goal of the code, and didn't feel a need to sketch anything at all.

Said one interviewee about problem five: *"Once you look at the code and figure out what it's doing, then it's like, okay, I can compute an average without writing it down, especially if it's only two values."* Another non-sketcher agreed, with the following thoughts on problem one: *"To me, it's very clear that you're looking for a minimum value within a certain range, and so for that, I didn't feel a need to necessarily write down anything."*

The opposite was also true. Describing their process for solving problem two, an interviewee recalled how not identifying a goal for the code led to a tracing

sketch: *“When I saw it I wasn’t like ‘Oh, I think I have a hunch that this code does this.’ I’m going to need to work through it.”*

3.4.1.2 Recognizing a pattern

In problems where an overall goal wasn’t initially identified, sometimes tracing through a portion of the code execution was enough to identify a pattern and confidently predict the final result. This approach could lead to halting sketching halfway though, as this interviewee described on problem two: *“After I sort of got the hang of it, I just started to skip through writing it.”* An interviewee who abandoned their trace on problem one said: *“You can also see the point where I realized like... and it clicked.”*

3.4.1.3 Goal and pattern recognition was not reliable

Despite the confidence expressed by many interviewees in their ability to recognize goals and patterns, they were often wrong. Consistent with the conclusions of Fitzgerald et al. [49], there was significant variety in how well students used the pattern-matching strategy.

3.4.2 Anticipated cognitive load

The complexity of the problem corresponded with the amount of effort put into sketching a trace, with more complex problems sparking more organized tracing approaches. An interviewee recounted their choice to only sketch a few values in an unorganized way as they were solving problem one: *“I know it’s a bit simpler, I immediately made like an assumption that it was easier, so I didn’t put as much work into making a table or a chart.”*

3.4.2.1 Complex variable interactions

How is complexity interpreted? Reflecting on their sketching choices for problem four, one interviewee felt that the amount of variable dependencies required a sketched trace: *“There were three [variables], I was like, “That’s too many,” and they’re all related. I was like, ‘I can’t remember that,’ and so I drew it out.”*

3.4.2.2 Arithmetic or other math-like cues

Math-like cues within code syntax also prompted a move to close tracing: *“This [code in problem 4] looks like an equation. All the other ones don’t look like equations. I guess when you don’t have really long variable names also that might help.”* Arithmetic was another factor that led to tracing: *“Instead of just reassigning values, I’m having to do some basic arithmetic, so keeping track of that through this method [a table] for me was the way to go about it.”*

3.4.3 Problem-solving progression: From goal search to tracing (and re-tracing)

Rather than immediate use of a tracing technique, interviewees typically described an initial search for code’s meaning, then a fall-back to tracing when no discernible goal or pattern was found.

Tracing was repeated two or more times, if an acceptable result was not found. Later tracing attempts were more organized, and more likely to take up a table-like format. In the words of one interviewee: *“When I do a problem, I usually just write stuff fast and try to get through to it. And then if I feel like that’s becoming difficult to keep track of everything, I’ll slow down and do something like this [a table], to make it easier to see.”* More organization was often assumed to bring a greater chance of success: *“I was kind of annoyed that I didn’t get the answer the first time so I was just like okay, putting down lines [creating a table to fill in]. I’m not going to get it wrong this time.”*

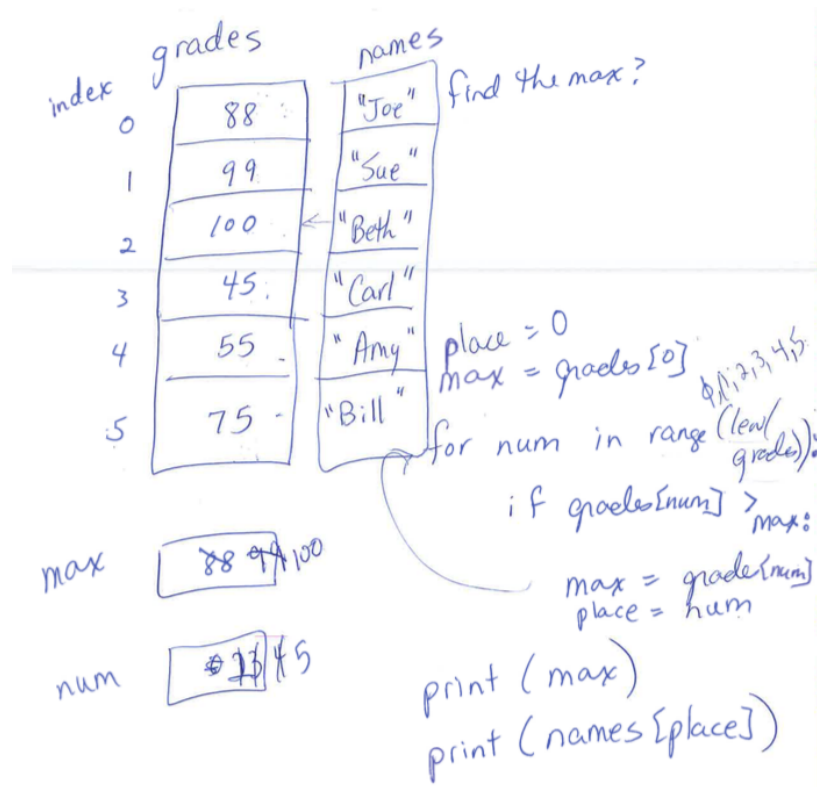


Figure 3.2: An in-class sketch by the instructor.

3.5 Why not sketch like the instructor?

The interviewees' instructor used sketched traces in class while teaching about lists and common list procedures, like finding a list's minimum value. One of her illustrations is shown in Figure 3.2. The instructor's choice to sketch boxes as representations of memory locations aligns with the design of the great majority of program animation software [141] (see Figure 3.1) as well as proposed designs for sketched memory diagrams (e.g. [152, 67]).

However, no participants re-created this sketching style while problem-solving. Instead, they opted for sketching techniques described in [88] and [33], where memory values are not bounded by boxes, but may be in rows, chunks, or tables. Why didn't students replicate the style repeatedly demonstrated by their instructor?

3.5.1 Seen as unnecessary and time-consuming

Interviewees described this sketching style as not worth the time and effort during exam conditions. *“Boxes take too long to draw, basically”*, said one participant. *“I’m always kind of hesitant to diagram things out to the max if I can save time by doing that.”* *“It would take me longer to (draw) the boxes than to write the number, and probably, it will be a mess,”* said another interviewee.

An interviewee described the instructor’s technique as overly detailed, and not prioritizing important values over unimportant ones: *“She’s trying to illustrate everything as opposed to me just trying to illustrate specifically what I need to get through a problem....she literally writes out every single thing. Even really simple things like if it’s a value and it’s fixed, she’ll write it. Like, it won’t change.”*

Others noted that they didn’t see a functional difference in the tracing sketch they performed and what the instructor sketched: *“I just didn’t circle the numbers, I guess.”*

3.5.2 Visual details seem distant from code

Some students didn’t feel that a box was the most appropriate representation for a variable. Said one interviewee who sketched traces using variable names and equals signs (e.g. $t = 1$): *“I think I just see it a little more clearly as saying that it’s a variable that’s equivalent to something, as opposed to maybe being a placeholder. I prefer to just write out what it actually is, in terms of a visualization.”* This student appears hesitant to move away from representations that aren’t similar to code syntax.

This prioritization of code was echoed by another participant: *“Because the high demand to put code in programming, I feel like, you don’t have time to sit there and draw a picture most of the time for a simple list.”*

3.5.3 Boxes are reserved for another purpose

For two interviewees, boxes had a clear meaning from their use in other subjects: identifying final results. Said one interviewee, *“If I cross them out, or box them then I end up getting very confused, because I use boxes and circles for other kinds of notation. I usually box my answers.”* Pointing at variables in Figure 3.2, another interviewee said: *“I’m guessing those are the initial values but it kind of makes it look like they’re the finals. Because from physics, like for homework, I always circle my finals....I saw the box first here and I’m like oh I don’t like that.”*

Boxing a key result, but not intermediate steps, is a technique students likely practiced across years of mathematics and physics courses. These participants didn’t want to change the meaning of this representation.

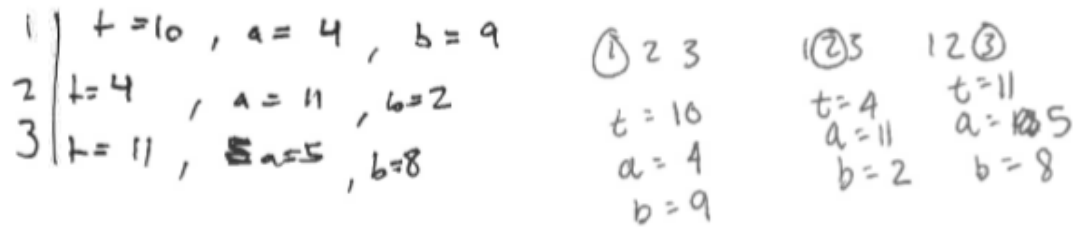
3.6 What do novice sketches include?

While participants’ sketches didn’t closely match their instructor’s sketches, or the style used in program animation tools, they did show a variety of consistent trends.

3.6.1 Organizing and structuring of traces

3.6.1.1 Organizing around loop iterations

Sketching out a trace was described as difficult without some sort of organizing structure. The index of the looping variable provided guidance during this cognitively challenging process. As one participant recalled: *“I think this one I had to actually do it a couple of times. The first time I went through I actually didn’t put the zero, one, two, and three, because that really helped me count the range. Before when I didn’t do that I really lost track of what my i value was so it got confusing, so I erased it a couple of times. Then I put the i values next to it, and it became much more clear.”*



(a) Each row of variable values begins with a loop index (b) Columns of variable values headed by loop indices

Figure 3.3: Tracing organized by loop indices

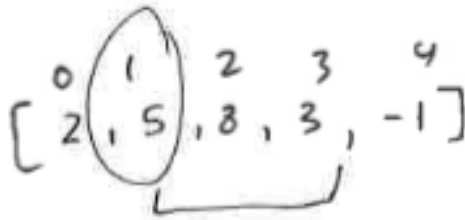
Describing their sketch (pictured in Figure 3.3a), another interviewee said: *“It’s just for me to establish here’s an iteration, here’s an iteration, here’s an iteration, just so I know what I’m dealing with and then not getting confused between iterations and numbers.”* This participant’s sketch indicates that loop indices are considered a distinct variable type. The loop variables aren’t labeled by their variable name, and instead serve to separate parts of the sketched trace.

3.6.1.2 Organizing around annotated lists

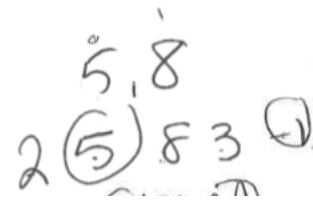
Rather than jumping to a full sketched trace on the questions involving lists, participants used the list itself to organize an abbreviated trace. An interviewee described the creation of such as sketch (pictured in Figure 3.4): *“And then I guess I circled five to start with that. And then I also labeled each value that I knew was probably going to change.”* Recognizing that the list is key to the question, the interviewee centered their problem-solving there, using the list to get an overview of what occurred in the code.

3.6.2 Persistence of past values and calculations

Interviewees were cognizant of the possibility that their sketched trace may need to be re-used or modified. Keeping previous values of variables around for



(a) Indice annotations, circling a key value, and demarcating a sub-list



(b) Circling a key value and rewriting sub-list

Figure 3.4: Tracing organized around a list

easy reference was a priority, as this interviewee described: *“if I mess up, I can go back and know that I either wrote a variable wrong or something like that.”*

For one participant, this need was enough to dictate their writing utensil: *“I like to do coding in pen because if I make a mistake I can clearly see it because I have to cross it. I can’t erase it, I have to cross it out.”* For another, this goal led to careful tallying of the evaluation of conditionals as well as variable values, *“so if I ever had to go back through a problem and check my work, I’d be like that’s why I did this.”* Figure 3.5b shows this sketch.

Keeping past values also allowed interviewees to better observe patterns over time. *“For me, I think it was just visual and then being able to see it laid out helped me also keep track of it. Because this is going back and forth and iterating it a bunch of times, so seeing the physical changes of the new variables was really helpful.”*

3.6.3 Anchoring with visible values and structures

3.6.3.1 Re-writing initial values

Where should students start when faced with a complex code tracing problem? Students often started with initial values. As one interviewee explained: *“I always rewrite the first list. It’s like for math. I always rewrite the original problem. I know some people don’t do that. It takes up a lot of time.”*

Despite the time commitment, several participants re-wrote starting values, like

1 2 3		
$t = 10$	$t = 4$	$t = 11$
$a = 1 + 3 = 4$	$a = 9 + 2 = 11$	$a = 3 + 2 = 5$
$b = 10 - 1 = 9$	$b = 4 - 2 = 2$	$b = 11 - 3 = 8$

(a) Intermediate values in arithmetic expressions are preserved

3 70 370 ✓	270 270	2 70 170
$7 = 7 ✓$	$4 < 5 ✓$	4
$c = 1$	$i2 = 1$	$4 < 2 X$
$i1 = 2$		$i1 = 1$
$i2 = 2$		

(b) Boolean expression values and variable values are preserved

Figure 3.5: Persistence of past values

the arguments passed into the function, or initial variable values.

3.6.3.2 Using code structure to anchor tracing

Some students re-wrote code in a way that oriented their trace, most often for looping structures. In Figure 3.6, this participant combined the for loop syntax with a trace across three iterations.

3.7 Discussion

Our results identify several problem-solving approaches of CS1 students that differ from the work of a strict “human compiler”.

3.7.1 Search for goals and patterns is primary

These findings suggest that the first task of novice code readers is an attempt to find patterns and goals within code. Novices scan code in an effort to recognize its functionality, or perform an incomplete trace in the hopes of seeing a trend. In both cases, the cognitively demanding work of full, close tracing is avoided.

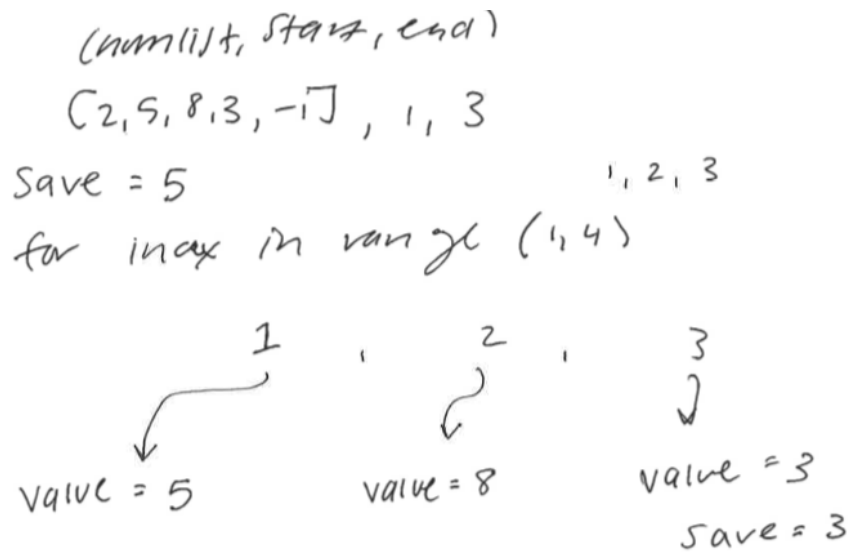


Figure 3.6: Organizing tracing around code structure

Such a strategy is understandable, considering the tedium of close tracing. If a student doesn't have a strong understanding of the notional machine, close tracing may be just as error-prone as looking for familiar code structures and divining the functionality from there.

Current visual program simulation approaches connect code structure (the syntax) to code behavior (the action of the notional machine). However, these approaches do not shed light on the function the code is trying to achieve. Can new approaches help learners not only better understand the notional machine, but also more accurately recognize patterns in code?

3.7.2 Variables are treated differently based on context

While a compiler doesn't treat loop variables differently than other variables, novice sketchers do. Loop variables are often used by novices to organize the tracing of other variables. Novices appear to have different classes of variables, based on semantic meaning.

Can visual program simulations treat loop variables differently than other variables? Or, can they use variable names to infer meaning, and emphasize or de-

emphasize variable changes accordingly?

3.7.3 Past values are retained

Novice tracers show a preference for retaining previous values as they trace, in order to identify patterns or be able to re-trace their steps.

In program visualizations, the previous value is typically replaced as soon as a new value is assigned to the variable. However, maintaining past values may allow novices to better detect patterns. What effects might retaining some past values in program visualization have on learners?

3.7.4 Variables are un-boxed

Novices are unlikely to maintain a box representation in their own sketched visualizations. The use of boxing may interfere with sketching strategies learned in other courses (like boxing the final value in physics). Or, it may simply be irrelevant when creating a sketch, but still useful when teaching. Visualizing variables in a table-like form creates a more direct opportunity for transfer to a sketching environment.

3.8 Limitations and threats to validity

Our small number of participants allowed for a detailed, qualitative investigation of novice programmer rationales for sketching. However, due to the small sample, we cannot claim any generalizability of our findings. Qualitative research explores experiences or artifacts in rich detail, establishing the existence of phenomena. In order to determine whether these behaviors are replicated by many students, we would need to increase our sample size.

While our findings suggest new possibilities for sketching and visualization

techniques, they do not translate into recommendations for classroom practice. We cannot claim that if instructors used the sketching approaches favored by students, better learning will result. Further research with more participants is needed to determine whether certain visualization techniques are typically associated with successful outcomes.

While our participants' sketches served to ground our interviews in past experience, our data ultimately depends on recall based on self-report. Additionally, in the weeks since the exam, participants likely improved their understanding of programming, creating additional challenges to accurate recall.

3.9 Conclusion

Sketching is a tangible trail of students' problem-solving process. However, pieces of paper alone can only tell us so much. By talking to novice programmers about their sketches, we identified their motivations for sketching differently than their instructor, for avoiding sketching and tracing, and for organizing their sketching in a variety of styles.

Visualization techniques in the computing education literature are designed by computing experts. Whether in program animation software or sketched by hand, these visualizations emphasize a high fidelity with the way the computer executes code "under the hood". The separation of memory values from code reflects the separation of instruction bytes from data bytes. The box analogy is consistent with parceling of only a certain number of bits per variable. The computer doesn't understand the semantic difference between a loop variable and a counter variable, so they are often illustrated identically.

Novice programmers aren't aware of the "reality" of the machines they are working with, and so while sketching and tracing they mix code and memory values, de-prioritize certain variables, and keep prior values around for reference.

Each of these actions supports their ultimate end: the search for goals and patterns in the code they are trying to understand at a human level.

Humans are different than compilers. While a computer must parse and execute code token by token, human learners can infer patterns and identify goals.

Our current visualization techniques don't offer affordances for this type of action. With inspiration from the visualizations novices choose to create for themselves, we may be able to create visualization techniques that students are not only more likely to use, but that could also support their ability to generate meaning from the code they are reading and tracing.

CHAPTER IV

“I’m not a computer”: How Identity Informs Value and Expectancy During a Programming Activity

Code tracing—simulating the way the computer executes a program—is a common teaching and assessment practice in introductory programming courses. In a laboratory experiment where code tracing was encouraged, I found that some struggling novice programmers described code tracing as not only cognitively complex, but also in opposition to their self-beliefs. One participant described himself as *not a computer*, and therefore unfit to execute code like the computer does. Another described himself as *not a programmer*, and did not value an activity that was only for learning about how code works. We mapped these learners’ self-narratives onto the Eccles Expectancy-Value Model of Achievement Choice to understand how identity relates to the choice to not trace code. While both participants valued what they could create with code, neither valued code tracing. Alternative activities might allow students with these identities to build skills in a way that aligns with their self-beliefs.

4.1 Introduction

In a series of programming epigrams, “founding father” of computer science Alan Perlis wrote *“To understand a program you must become both the machine and the program”* [115]. Instructional approaches in many programming classrooms echo this sentiment, focusing on code tracing [140] as a method to improve programming skills. Code tracing involves simulating the execution of a program: describing the order in which lines of code run, which values are created and modified, and when the program starts and stops.

Code tracing is an authentic practice for future programmers. Software developers spend large amounts of time reading existing code, rather than writing code from scratch [9]. Tracing through an example can aid in debugging code [103]. It can also help a programmer describe what a program does in “plain English” [86]. In classroom settings, the ability to trace code unaided is correlated with the ability to write code and the ability to describe the purpose of code in natural language [93]. Hierarchies of programming skills describe code tracing as a primary skill that students naturally learn [86] or should be taught [163] before they learn to write code or explain the purpose of code.

However, code tracing by hand is difficult, tedious, and novice programmers don’t always do it, even when it is a helpful strategy [88, 33]. Learners describe the cognitive demands of tracing as one reason they try to avoid it by using alternative methods, like looking for familiar code patterns [34]. Code tracing is also typically removed from any larger context of what the code will be used for. Computing education researchers have advocated for code tracing examples that use unusual code structures, so that learners are not “distracted” or “influenced” by context clues like meaningful variable names [88, 110].

Code tracing studies are typically performed with novice programmers in traditional programming courses, housed in computer science departments. Hierar-

chies of programming skills developed from these studies do not consider how different students may have different values for different programming tasks. Future computer scientists likely value a deep understanding of the operation of programming languages. End user programmers and data scientists use code when needed to achieve their own goals, and may not value knowing how the code actually works. In this study, I explore: How do non-computer science majors relate their value for code tracing to their identity?

In a laboratory setting, I asked undergraduate and graduate students with some prior programming experience to complete problems designed to encourage and isolate tracing behavior. To our surprise, some participants pushed back against instructions to trace code. Beyond avoiding tracing due to its difficulty, these participants described themselves as not the “type of people” who wanted to understand code tracing. I analyzed the attitudes, values, and beliefs that these participants expressed, and mapped them onto the Eccles expectancy-value model of activity choice in order to understand how identity influences the choice to trace code.

4.2 Background

4.2.1 Identity, programming, and applications

In the United States, students who choose to study programming in college are predominately White and Asian males [22]. Explanations for this phenomenon often suggest that women and underrepresented minority groups are less likely to see the culture of computing as aligning with their values. The efforts by Jane Margolis and her colleagues to understand the low representation of female students Carnegie Mellon University’s computer science department found a difference in the values men and women had for computing. Women were nearly five times as

likely to link their interest in computing to application areas, like medicine [97]. Male students, on the other hand, more often expressed views like “it is the code itself that is interesting, even more so than the actual effect it has” (pg. 53).

A similar difference in motivation for understanding programming seems to be present between professional software developers and professional web and graphic designers. Dorn and Guzdial [36] found that web and graphic designers had often taken traditional introductory programming courses, but found them frustratingly focused on programming language syntax rather than applications. Increasingly, jobs beyond professional software developers require programming, in areas as diverse as business, science, and art [18]. Interdisciplinary college departments, such as “iSchools” [94], cater to this demand for data science and other applied computing skills.

4.2.2 The Eccles expectancy-value model of achievement choice

Drawing on prior work about decision-making and goal achievement, Jacquelynn Eccles and her colleagues proposed that two factors most directly influence the choice to select and complete a task: (1) a student’s expectation of success and (2) the value a student has for the task [43]. Many other factors influence these two primary factors, such as the student’s personal and social identities, goals, and their memories of previous achievement. The interaction between these factors is described in Eccles’ expectancy-value model of achievement choice, which provides a framework for a sociocultural analysis of motivation and activity choices in educational settings [43].

Four factors contribute to the value that a student has for a task, which Eccles terms Subjective Task Value: (a) attainment value, when the task aligns with the student’s self-image; (b) interest-enjoyment value, when the student expects to enjoy the task; (c) utility value, when completing the task helps the student achieve

a goal; (d) relative cost, when task requires time and loss of other opportunities, decreasing its value [43].

4.2.3 Tracing code to solve problems

Code tracing involves simulating the running of a program: describing the order in which lines execute, which values are created and modified, and when the program starts and stops. It's a careful walk through the mechanisms of how a program works inside the computer [140]. Dozens of program visualization tools have been built to illustrate code tracing to new programmers, [141]. These systems are typically similar to the visual debuggers found in professional integrated development environments. Instructors and students often trace code by hand, on whiteboards or paper. Teaching that emphasizes tracing seems to result in learning benefits on later exams that include tracing problems [164].

Tracing can help with other tasks. During microgenetic studies of novice programmers, Lister and colleagues noticed that novices sometimes traced through the execution of a concrete example in order to inform their explanation of the purpose of a program [86]. Viewing program visualization tools is associated with improved debugging ability [103].

While tracing studies often describe novices' cognitive processes, they do not address learners' motivations or affective state while solving these problems. To our knowledge, ours is the first study that investigates how students' self-beliefs, values, and identity is related to the task of code tracing.

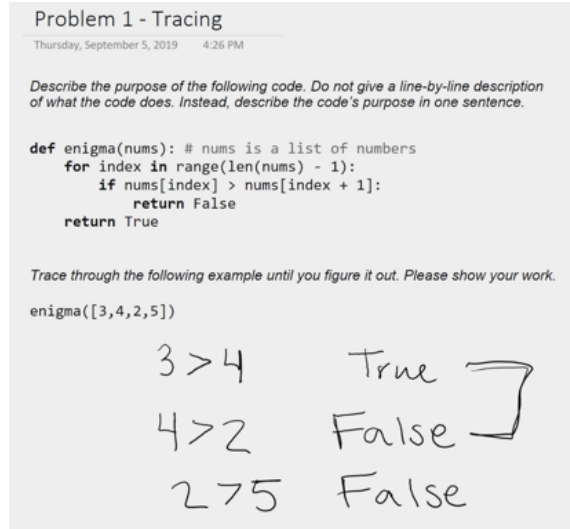


Figure 4.1: A participant's work on Problem 1, during the tracing stage.

4.3 Method

4.3.1 Task

I asked participants to complete an “explain in plain English” task [108], using tracing as a support for problem-solving. In the initial reading stage of the task, participants were shown a short program, displayed for 30 seconds on a tablet, and asked to give their best guess of the code's purpose. If the response was correct, a more challenging problem was chosen. If the response was incorrect or incomplete, participants moved on to the tracing stage. A function call was added to the tablet display, and participants were asked to trace the execution of the function call, using the tablet's pen. Participants were told to stop tracing and let interviewers know once they had determined the program's purpose.

4.3.2 Problems

Code segments were selected from those used in “Explain in plain English” studies [89, 93, 108] and in sketched tracing studies [34]. The code was translated into Python functions. All participants saw the same initial problem. A second

problem was dynamically chosen based on the participant's difficulty with the first problem. If time allowed, third and fourth problems were also chosen.

4.3.3 Participants

I recruited novice programmers from undergraduate and graduate students at a large public research university in the Midwestern United States. Students were eligible to participate if they had completed 1-2 formal programming courses at the college level. Participants were recruited during the 1st, 2nd, and 3rd weeks of a 15 week semester, primarily from the university's iSchool. Recruiting included flyers placed across campus, announcements sent to iSchool programming courses, and classroom visits to iSchool programming courses. Participants received \$10 for participating in the study. I conducted the study with 12 participants, 8 undergraduates and 4 PhD students.

4.3.4 Interview Protocol

I conducted 30 minute interviews with participants during the 2nd, 3rd, and 4th weeks of the semester. Each interview consisted of 2-4 purpose-finding tasks, as described above. Two interviewers (the first and second authors) were present at each interview. A "cheatsheet" describing the operation of Python syntax structures was available to participants. After each task, I asked our participants semi-structured reflective questions about their problem-solving process. These questions asked participants to describe the moment they understood the code's purpose and what happened leading up to that realization, to explain what they found helpful in completing the task, and to share what they had determined in the initial reading stage, before tracing. In the final task of each interview, I asked participants to talk aloud, describing their thoughts while they traced.

Sometimes, lack of knowledge about how Python works precluded participants

from being able to accurately trace. Interviewers answered any questions participants had about the operation of Python syntax structures (e.g. loops, arrays) and built-in functions (e.g. `range()`, `len()`). When a participant was clearly struggling due to a lack of knowledge about the mechanisms of Python, interviewers intervened and provided a relevant example and explanation.

4.3.5 Analysis

The interviews were audio recorded and transcribed, and activity on the tablet was screen recorded. Inspired by participants' unexpected talk about the value of the tracing task, I analyzed the transcripts to highlight the ways participants described their identity and motivations. I used Values Coding [117, 126], supplemented by In Vivo Coding [25, 126]. Values Coding highlights the belief system of the participants by identifying their values (the importance of people, things or ideas), attitudes (the way participants think and feel about people, things, or ideas), and beliefs (rules and interpretations) in their talk. In some cases, attitudes, values, and beliefs could be expressed in the participant's own words, so I used In Vivo Coding to preserve participants' tone and voice.

The first author coded all interviews while the second author coded 40% of the interviews, including the case studies described below. During weekly meetings across the course of a month, I discussed our codes, developed themes, and identified trends across participants. Using reflective summaries as an analysis tool, I mapped our themes onto the Eccles Expectancy Value Model of Activity Choice [43].

4.4 Case Studies

While the interviewers did not ask questions about learners' identities, motivations, or values in the interview protocol, some participants used the interview as an opportunity to share their judgements of the task. Two participants described most fully their perspectives on their own ability, as well as the reasons for their low value of the experiment's tasks.

4.4.1 Charles: I'm not a computer

4.4.1.1 Code reading is confusing, and I won't improve

Reflective questions about Charles's problem-solving during the first problem of the interview prompted a negative response. "I really hate—I guess I shouldn't say hate. But I really dislike like, how convoluted like parameters can get inside a code. And I recognize that they're very important, but that's like so counter everything else you do in life," he said. He continued to describe the task of understanding each small piece of code "the hardest thing for me with code in general."

Overlooking some parts of the code was also a roadblock. While explaining a mistake he made on the second problem, Charles said, "I have a tendency to do that (laughs) with code. I like to jump to a conclusion and not have any alternative." When he didn't notice an important code structure while reading the final problem, Charles said that he would never read code accurately: "If we were to do 10 of these, I'm sure each one of them I would look over a small component of the code."

Charles connected his struggles with code tracing in the experiment to difficulty in his prior programming course: "I guess like, in general, I found that like, the least helpful thing in my programming class was reading code, as weird as that sounds. I always felt like I walked out of the class feeling like I knew less than

I did going into the class.” Charles finds code reading impenetrable, dislikes the task, and has no confidence that he can do better than his current efforts.

4.4.1.2 Why not just execute the code?

Charles believed he could solve problems like the ones in the experiment, if he could only execute the code on a computer: “I feel like no matter how much I do it, it doesn’t really help me, uh, learn what the code’s actually doing until I execute it.” He described tracing activities on past exams as “demoralizing”, “because I can’t execute it, so I can’t see what it’s doing. And since I don’t know, I just feel like I don’t know and I can’t work through the problem and like, try and solve it.” Without the capability to execute code, Charles has very low confidence in his ability to understand code.

Charles elaborated on how inauthentic he felt it was to be tested on code tracing. “Nowhere outside, I feel like, a college setting is ever gonna ask you that question,” he said. “Who writes code on a whiteboard and then tries to solve it in their head?” In his view, there is no need for tracing knowledge when he could simply run the code on a computer: “It always seems like a really strange way to try and teach someone code when like you could just execute it and see where it goes with that like hands-on component.” Charles views code reading tasks as unrealistic and a “weird” thing to be tested on, at least partly because there is no use of the computer.

As Charles reflected, he mentioned that code tracing might be relevant for code writing, saying, “ultimately, I’m never gonna call upon this knowledge again, other than when I’m writing code.” However, he qualified that statement by describing how he could probably complete a similar code writing task, even though he wasn’t able to complete the code tracing task. He described a writing code task as “more beneficial” and claims “I would probably much more likely get to this

result.” By contrast with tracing, Charles views writing code as an easier task to complete, and more helpful to his development.

4.4.1.3 It’s great that we don’t do this type of work in the iSchool

At the end of the interview, Charles connected his beliefs about his ability and the authenticity of the task with his choice of major: “Which is why I really like [the iSchool]. ‘Cause I feel like there’s much more hands-on components, rather than, like, me just being tested over and over and over about things that I don’t quite understand.” He feels that his iSchool course allows him more opportunities to make use of the computer, rather than the code tracing tasks he despised from his prior course.

Charles summarized his perspective on code tracing with an analogy, saying, “Yeah, I mean, it’s just like...it makes me think like a computer. But I’m not a computer. And it’s not that I can’t work with the computer in tandem. I mean, that’s why we have the computers.” Tracing is simulating the computer, and Charles is fixed in his belief that he cannot think like a computer thinks. Computers read every detail of code and execute it without error. Charles does not believe he can do that, but he doesn’t count himself out of being someone who writes code. He can use the computer as a tool and complement the computer’s strengths with his own strengths.

4.4.2 Luke: I’m not a programmer

Luke is a PhD student in the iSchool who uses programming in his research. His prior programming experience included a formal programming course approximately three years in the past, as well as applied experience using code to create products used in research activities.

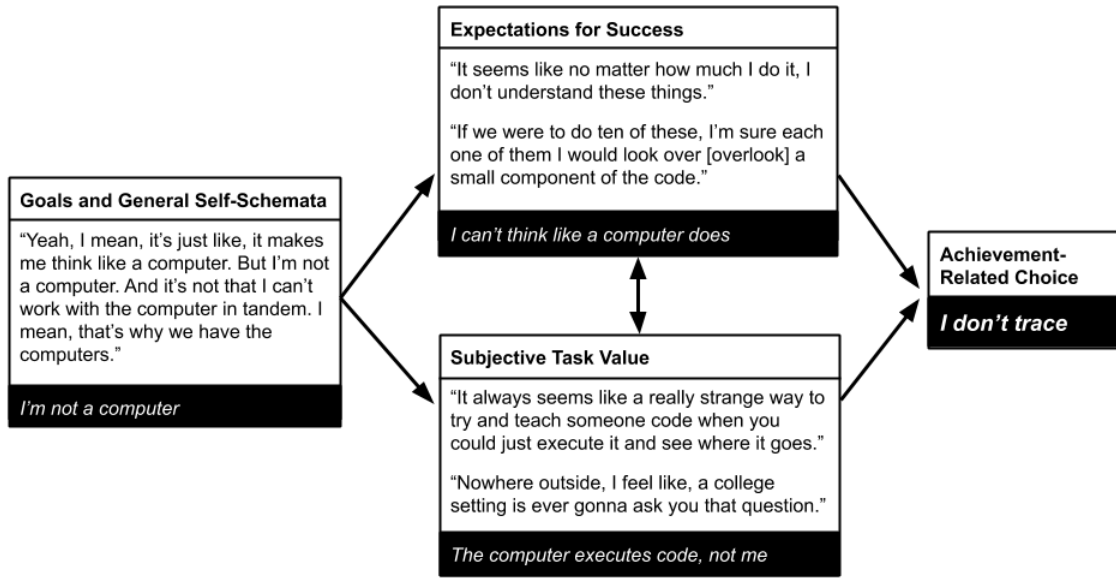


Figure 4.2: Charles' self-narrative mapped onto relevant aspects of the Eccles Expectancy-Value Model of Achievement Choice [43].

4.4.2.1 I don't remember this type of stuff

When I showed Luke the first problem, his negative reaction was immediate. "I hated this sort of work honestly," he stated, looking at the code on the tablet. During the tracing stage of the first problem, he didn't draw anything, and noted that he didn't remember many details of how the code worked. He said he would have to review basic Python in order to complete the problem, because "the way that I, I learnt everything in the past, was like 'Okay. If I don't need it anymore, it's done.'" Interviewers provided help with the relevant Python semantics at this point, but Luke was reluctant to engage and keep trying to trace.

When asked later about why he had expressed such a negative reaction at the beginning of the interview, Luke responded that "I have to force myself to think extra about something I know I'm not going to use." The tracing problems required a high amount of cognitive effort, but no clear payoff. Luke further described the problems as being "like a crossword puzzle that's not fun." He disliked problem-

solving that didn't solve a problem he was interested in.

4.4.2.2 The only point of these problems is to learn the language

In the reading stage of the third problem, Luke described how the code would work line-by-line for a sample input he devised. When pressed to describe the purpose of the code in a single sentence, Luke deadpanned, "Um, teaching people Python." This meta-commentary continued later in the interview, when he described his past experience with similar tasks: "this sort of stuff was blatantly to write Python." There was no larger goal for these tracing tasks; they didn't contribute to anything except knowledge that Luke didn't value. He continued, "There's times I'm just like 'Why would I ever use this?'"

The interviewers asked Luke to reflect on whether the code in the tracing task was similar to other code he worked with. He didn't see many similarities. Describing his past coding experience, he said, "I just liked to design something, design an app, design some sort of robot personal interaction, to design like, you know, um, just like JavaScript as well, just like something that someone would go through online for a certain experiment. So like I never really did this type of like work, you know?"

4.4.2.3 I use code to achieve goals I care about

By contrast with the tracing tasks in the interview, Luke characterized the code used in his design work as "more fun". The reason seemed to be that coding in that context helped him build something he valued: "Whatever it is that I agree with, the design itself, like, so the code is just a means to an end, of creating an interaction, or creating a product or creating whatever else, right? So it's like one component that gets you there." Writing code is only useful for something else; the code itself and how it works is not valuable. Luke said, "I mean the purpose itself

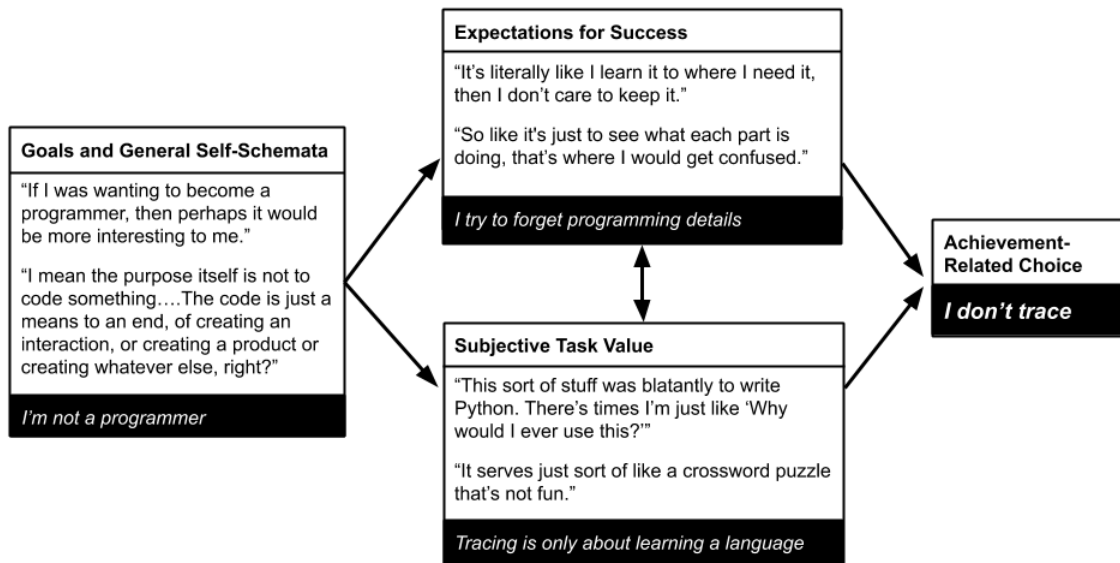


Figure 4.3: Luke’s self-narrative mapped onto relevant aspects of the Eccles Expectancy-Value Model of Achievement Choice [43].

is not to code something.” Instead, “the purpose would be like, research through design or to create some sort of interaction or to iterate on something.” Coding might be necessary for certain projects, but coding isn’t the point.

Finally, Luke reflected on his own prior programming course and the fact that he didn’t enjoy it. He chalked part of the reason up to his identity. “I mean for one, if I was wanting to become a programmer, then perhaps it would be more interesting to me.” He applauded recent changes at his undergraduate university that created distinct introductory programming courses for different majors “to be more applicable to students’ interests.”

4.5 Discussion

Both Charles and Luke expressed a strong negative opinion about our tracing tasks as soon as they attempted their first problem. As the interview continued, they provided well-developed descriptions about the reasons they found tracing

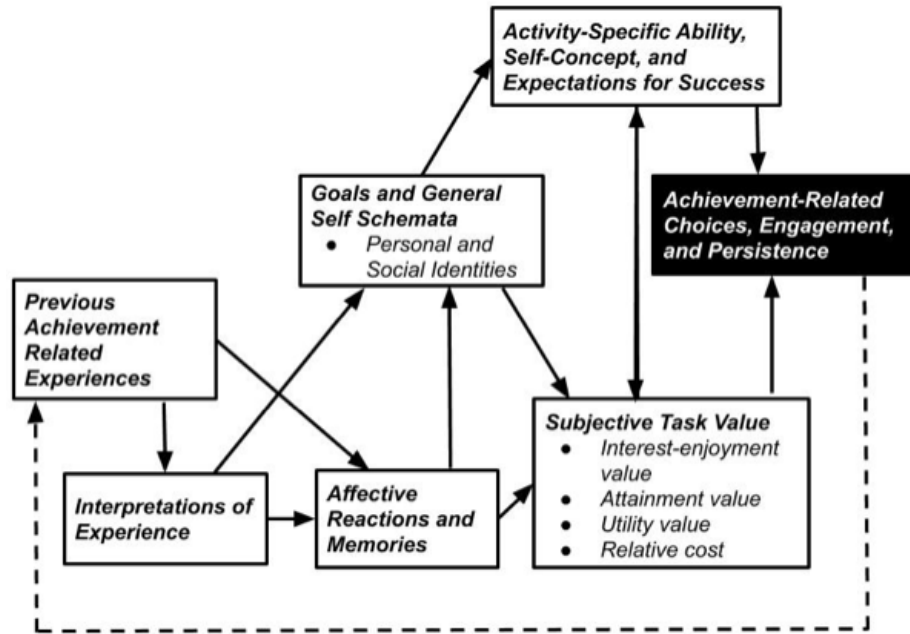


Figure 4.4: Relevant components of the Eccles expectancy-value model of achievement choice (modified from Eccles [43]). Cultural Milieu, Socializer’s Beliefs and Behaviors, Stable Child Characteristics, and Perception of Stereotypes and Socializer’s Beliefs are not represented.

tasks both difficult and not valuable. Mapping their values, attitudes, and beliefs onto elements of the Eccles expectancy-value model of achievement choice (see Figure 2) creates a clearer picture of how the different parts of these learners’ self-narratives interact and drive their opposition to code tracing tasks.

4.5.1 Previous achievement-related experiences, Interpretations, and Affective reactions and memories

Luke and Charles disliked their introductory programming course. Charles describes his affective memories in detail, sharing feelings of helplessness and being demoralized by code reading activities in exams and in class. His interpretation and memories of repeated failure likely influenced his self-beliefs about tracing ability.

4.5.2 Goals and general self schemata

Both participants summarized their personal identities in opposition to another identity. While Luke is clear that he is not a programmer, Charles emphasizes that he is not a computer. Both Luke and Charles define themselves at a distance from people who are thinking deeply about how code works. However, from this distance, both participants see themselves as someone who uses programming. Charles is someone who can work with the computer, and Luke is someone who builds things with code.

4.5.3 Activity-specific ability, self-concept, and expectations for success

While both Charles and Luke expressed low expectations of success during the experiment's tracing tasks, they prescribed different reasons for their lack of code tracing skills. Charles felt that he inherently did not have the ability to remember the details of code tracing, or observe all the details of a program. Computers can accomplish these tasks easily, however Charles is not a computer, and cannot trace code. On the other hand, Luke described an almost conscious forgetting process of programming knowledge that was "no longer needed" to achieve his immediate goals. However, he expressed confidence in being able to recover those skills through study. Luke no longer needs this knowledge because he is not a programmer.

4.5.4 Subjective task value: Interest-enjoyment value

Luke described his lack of enjoyment in more detail than Charles. While it's fun for Luke to build things, it's not fun to work through the details of how the code works. Code tracing is as detail-oriented and demanding as a crossword puzzle, but less fun than that pastime, and just as disconnected from any larger goal. For Charles, the work of code tracing seemed to recall negative memories of frustrating

and unenjoyable prior experiences in class.

4.5.5 Subjective task value: Attainment value

People have attainment value for a task when the task aligns with their self-identity. In both Luke and Charles' descriptions for why they don't value code tracing, they described the task as in opposition to their identity. Tracing is something that a programmer values, because it helps them understand how a programming language works, but Luke is not a programmer. Tracing is something the computer does when it executes code, but Charles is not a computer.

4.5.6 Subjective task value: Utility value

People have utility value for a task when completing that task helps them achieve their goals. Charles does value understanding code, at least as much as it can help him write code. However, Charles does not believe that tracing helps him understand, and in fact makes him feel like he knows less. While Charles mentioned that tracing knowledge may be helpful for writing code, he also says writing code directly would be a more beneficial task than pure tracing practice. Luke expressed goals of creating products that do things he cares about, but didn't see tracing as relevant for that activity. Neither Luke nor Charles described any utility of code tracing for explaining, modifying, or debugging code.

4.5.7 Subjective task value: Relative cost

For Luke, it is quite time-consuming to look up relevant knowledge for code tracing, since he doesn't remember much tracing knowledge. As someone who forgets things when they are "no longer needed", recalling code tracing details is always costly. Luke does believe it's possible to refresh this knowledge, but it would take focused study. For Charles, the cost of code tracing is even higher than

for Luke. As someone who just doesn't notice details of code, it's unclear he will ever be able to accurately trace code. With fixed beliefs about his ability, the cost seems infinite.

4.6 Conclusion

While previous research has focused on the ways that high cognitive load may influence the choice to trace code, our analysis identifies relationships between identity, expectations of success, and value for code tracing. I describe two identities that relate to choices about code tracing: *I'm not a computer* and *I'm not a programmer*. In our case studies, 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. Both Luke and Charles define themselves at a distance from people who are thinking deeply about how code works. However, from this distance, both participants see themselves as someone who uses programming. Charles is someone who can work with the computer, and Luke is someone who builds things with code.

Tracing code can certainly be difficult, intricate, and removed from the context of code writing. However, it is commonly positioned by computing education researchers as a "gateway" skill to programming expertise, frequently occupying the earliest rung in theorized pathways of programming learning [86, 163]. For learners with the identities I describe, this code tracing "gate" is closed. Tracing is already so difficult that many computer science majors struggle with it [88]. For learners who do not value understanding the mechanisms of code and intricacies of a language, there is little motivation to put the required effort into code tracing. If you see code as only a means to an end, why learn about code that doesn't

have an application? If you want to work with the computer rather than be the computer, why simulate the machine?

Our learners are using code to achieve their goals, but they reject one of the most common activities in programming classrooms. To meet this type of programming learner where they are, I propose an exploration of programming learning activities that are function-oriented, contextualized, and authentic. While some programmers may not ever fully “become” the program or the machine, they can start by investigating what the program and the machine does for them.

CHAPTER V

Defining, Building, and Evaluating Purpose-First Programming

5.1 Introduction

Enrollment in undergraduate computing courses is undergoing exponential growth, the majority of which is driven by students from majors besides computer science [21]. Non-computer science majors often want to learn programming for reasons other than as preparation for a career in software development. One intention is to become a “conversational programmer”, someone who knows enough about technical topics to communicate with co-workers, but doesn’t often program in their work [28, 29]. Another is to become an “end-user programmer”, who programs to achieve goals in their domain, but doesn’t build software as a product [79].

For students targeting the wide variety of careers that involve programming but aren’t software development, such as user experience design, product management, business analysis, data science, and entrepreneurship, programming is a tool to be used for a specific need. However, typical introductory programming instruction focuses on the features of a programming language, not the applications of code in a domain [2]. Many theories of programming instruction recommend

an early focus on a language's semantics, often in the context of "toy" problems, reasoning that such content will prepare learners to have a deep understanding of programs in any domain [86, 163, 140].

This approach doesn't work for all learners. After trying formal and informal methods to learn programming, conversational programmers report a lack of benefit and feelings of failure [157]. They found programming learning resources didn't align with their needs: instruction focused too much on syntax and logic, and not enough on how to apply code to solve problems [157]. End-user programmers have similar concerns. Web designers disliked typical programming courses, viewing them as focused on syntax rather than concepts [36].

In my work, I've found that code tracing [140] in particular can be a source of cognitive and motivational challenge for novice programmers. Closely tracking memory values requires high cognitive load, which can lead to a low expectation of success on tracing tasks [32]. The high cognitive load prompts some learners to avoid tracing and search for code's goals and patterns instead [34]. To counter this tendency, the programs novices are expected to trace often lack a meaningful purpose or pattern, or are even designed against common practice to subvert learners' expectations [88]. As a result, tracing problems are inauthentic and inapplicable, and have low value for those who care about what code can achieve, not how it works [32].

After learning programming in the typical manner that prioritizes programming language semantics, novices are stuck in a "Turing tarpit", where *"everything is possible, but nothing of interest is easy"* [115]. These learners may understand how loops, selection statements, and functions operate, but putting those elements together to understand or write programs similar to those of a professional is a long way off. Conversational programmers and end-user programmers value the application of programming in their domain, and currently undergo a difficult and

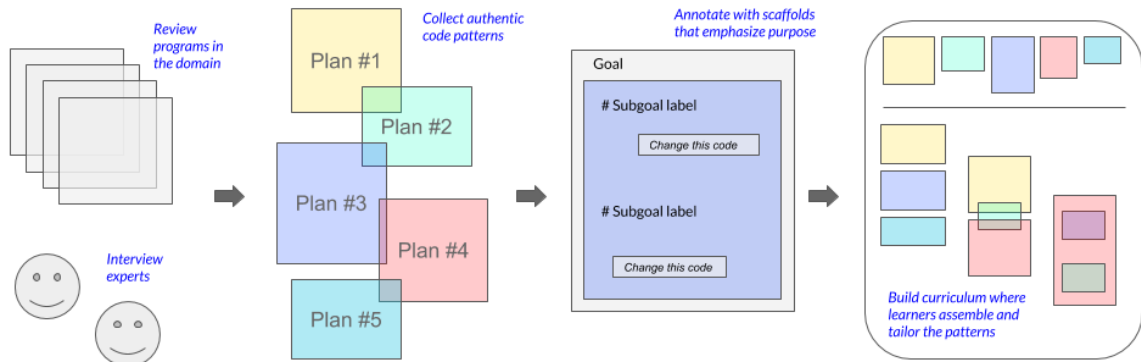


Figure 5.1: The development of a purpose-first programming module

demotivating slog through the tar-pit before they can see any such application. Learning about programming with a clearer, easier, and more immediate connection to code’s purpose may be more motivating for this population, resulting in a longer engagement with programming learning.

New technologies are needed to create programming learning activities where novice programmers quickly and easily create or understand authentic and meaningful code. In this chapter, I describe the development and evaluation of a technology-supported approach to programming learning that allows learners to engage with information about code’s purpose.

5.1.1 Summary of contributions

5.1.1.1 I investigated the response of a variety of novice programmers to purpose-oriented support

In a series of four focus groups and a survey, I found initial support for scaffolding that emphasized code’s purpose among novices who didn’t plan to become software engineers. I found that conversational programmers in particular appreciate extra support when learning to program, and value general understanding over a focus on detail.

5.1.1.2 I outlined a novel learning approach that emphasizes code’s purpose

In order to cater to these needs, I introduce **purpose-first programming**, a brief, authentic, and purpose-driven learning approach designed to motivate novice programmers who care more about code’s applications than its semantics. By focusing learning on a small number of common patterns in a domain-specific context, learners are supported in quickly creating code that reflects expert practice. Scaffolds (assistive mechanisms [156, 162, 68]) provide guidance as learners write, debug, and explain code by emphasizing the purpose of code patterns and ways they can be modified.

5.1.1.3 I designed a proof-of-concept curriculum, implementing the approach

To investigate the effectiveness of purpose-first programming for motivating conversational and end-user programmers, I develop a *proof-of-concept* purpose-first programming curriculum that teaches five patterns in the domain of web scraping. This curriculum provides information, structures, and feedback that highlights how code achieves goals in this domain.

5.1.1.4 I ran a lab study to evaluate novices’ motivation for and use of the approach

I evaluated the curriculum with nine novice programmers, including five conversational programmers and four learners who want to program in their jobs. Purpose-first programming learning enabled novice conversational programmers to complete scaffolded code writing, debugging, and explaining activities after only a half hour of instruction. I found that learning with purpose-first programming is motivating because it engenders a feeling of success and aligns with these learners’ goals and identities.

5.2 Motivation for the approach

5.2.1 Conversational programmers and end-user programmers want to understand the purpose of complex code, but also want to avoid detailed semantics

Novice programmers who want to learn code to understand its applications, such as conversational programmers and end-user programmers, have learning motivations that are seemingly in conflict. They want to understand or write code that can achieve real-world tasks, like data analysis or web design. Such code is often quite complex. At the same time, these learners want to avoid a close study of syntax and execution flow in favor of conceptual and application-focused understanding [157, 37].

In addition, since conversational programmers plan to work with developers, they value learning code that is authentic to the work of real programmers [28, 157]. For instance, conversational programmer undergraduates preferred to learn a more challenging industry-level language (Java) than a language designed for non-programmers (Processing[58]), perceiving Java as more useful, practical, and marketable.

How can these learners understand or create complex and authentic code without a deep dive into code tracing? Theories of programming instruction propose that knowledge of code semantics can help novices understand the purpose of programs [163, 86], but these programming learners reject learning experiences that focus on syntax and logic [32, 36, 157].

The Structure Behavior Function framework [35, 55] tells us that some knowledge of code behavior (and understanding of "how code works") is necessary in order to design and understand novel programs. To avoid syntax and semantics while understanding meaningful and useful programs, these learners will need an

alternative method of conceptualizing code behavior.

5.2.2 Programming tools for non-developers often avoid industry- standard code, so they don't provide disciplinary authenticity

A wide variety of technologies have been developed to make programming more accessible to non-programmers. Such systems often involve block-based or other visual interfaces that make the process of programming easier by reducing the potential for errors. Block-based languages like Scratch [96, 95] eliminate the need to remember syntax since the jigsaw-like shapes for commands and functions fit together in an intuitive way. Snap! [66] is another example of a block-based programming language, with more features than Scratch, including first class functions and support for multimedia. Helena [26], a block-based language for web scraping tasks, has helped sociologists, engineers, and public policy researchers obtain data from websites.

A community of practice perspective [85] can explain why these tools don't meet the needs of aspiring conversational programmers. Lave and Wenger described how desire for learning is motivated by the alignment of learning activities with the tasks performed by people in the communities learners want to join [85]. Sociologists value data collection, and so learning Helena is relevant to the practices of sociologists. Conversational programmers, on the other hand, want to understand tasks that software developers complete [28, 29, 157]. Weintrop showed that even though Snap! allowed learners to use advanced programming concepts like first class procedures, high school students doubted its authenticity since text-based languages are standard in the software industry [158]. Use of industry-standard languages and text-based programming may demonstrate to conversational programmers and other novices that what they are learning is authentic to their future careers.

5.2.3 Plans may be a more motivating way for conversational programmers to think about code

Soloway and his students used schema theory to identify *programming plans* in the 1980's [138, 137, 133]. Plans are chunks of code that achieve particular goals, like guarding against erroneous data or summing across a collection. There is evidence that both novice and expert programmers think about code in terms of plans [136]. Student errors can be explained in terms of misunderstanding plans [135, 146] and errors in composing plans [144, 145]. Over time, error rates on plans decrease with practice [143], while error rates on syntactic structures does not [123], which suggests that plans better describe how students learn programming than syntax structures.

According to Expectancy-Value theory, motivation for an activity is explained by expectancy of success in the activity and subjective value for the activity [41]. Conversational programmers and end-user programmers found that existing instructional materials typically focus on lower-level details like syntax and execution flow [157, 36]. Closely tracking code's execution can result in a high cognitive load [34], resulting in a low expectancy of success for some learners [32]. When thinking about code in "chunks" [53] of programming plans, the difficulty of understanding code may be reduced, increasing the expectation of success.

Activities designed to help learners understand syntax and execution flow typically involve problems intentionally stripped of context, so learners can focus on code semantics "without distraction" [88]. For conversational programmers and end-user programmers, such problems have low utility value [43] because the connection to activities in the workplace is unclear [157]. By contrast, programming plans associate a code pattern with a goal relevant to the user, making the purpose of code evident. Domain-specific code plans have an even clearer connection between code and application, potentially resulting in a high value for plan-based

learning.

Students programming in terms of high-level, domain-specific code plans have successfully solved programming problems that students programming in more traditional languages have not. The Rainfall Problem has been challenging students for over 30 years, with most studies finding that less than 20% of students are able to solve it successfully [130]. Fisler found that she could reliably get most of her students to successfully solve the Rainfall Problem by using a high-level functional programming language [46]. Fisler explains that students were able to succeed because they mapped the high-level functions to domain-specific plans that were easily composed by the students into a successful solution [47].

5.2.4 Other systems have provided plan- or example-based support

The GPCeditor was a programming tool for students to support them in learning Pascal programming through the specification of goals and plans [64]. Students learned plans which they then transferred into a more traditional Pascal IDE. SODA extended the GPCeditor with support for program design and software engineering practices [73]. The GPCeditor and SODA focused on traditional introductory programming learning, rather than supporting programming learning in the context of a domain.

Emile provided adaptable scaffolding for building physics simulations through HyperTalk programs [61]. Students in Emile constructed programs out of plans with *slots* which allowed for specifying a plan for a particular purpose (e.g., to generate accelerated motion for a given velocity and acceleration source for a given object) through a supported process. Students still did not have to learn the language syntax and semantics to be able to achieve their purpose (the construction of physics simulations). Students in Emile did learn physics, which suggests that a plan-based approach to learning programming can lead to learning within the

purpose domain.

There is a long history of providing purpose-oriented support in the form of examples or cases. The *minimal manual* approach of Carroll and others [24] is built around providing examples of the processes for common tasks. The examples are indexed by task, like “how I delete text in my document?” or “how do I add a button to my screen?” Evaluations of minimal manuals show that they are successful (in terms of user productivity and satisfaction), and are most successful when steps have to be inferred by the user [13]. Minimal manuals have been used successfully to help non-professional programmers succeed at programming tasks [6].

5.3 Formative study: Investigating the responses of novice programmers to purpose-oriented assistance

Providing additional purpose-oriented support during programming learning appears reasonable in theory. However, will novice programmers, particularly those who don’t plan to become professional programmers, see this approach as appropriate for their needs? To understand the responses of learners to early prototypes of the purpose-first programming approach, I performed a series of focus groups and a follow-up survey with a diverse set of students taking a data-oriented programming course taught by the School of Information at the University of Michigan.

5.3.1 Focus group results

Across four focus groups, I talked to eleven students (eight female, three male) about their goals with programming and what learning approaches they felt best prepared them for their future. I provided prototypes of purpose-first programming activities as probes [14] to gather feedback. The focus groups lasted 30 min-

utes, and participants received a \$25 Amazon gift card. Transcripts of the focus groups were analyzed to create "personas" [31] representative of students' goals and preferences. I developed two personas: the *conversational programmer* and the *analyst*.

5.3.1.1 Connie the conversational programmer

Connie is interested in a career in "user experience design" (P8), "project management" (P11), or "digital strategy" (P9). She wants to be "informed about coding" (P8) in order to "understand what the coders are doing and what they can and can't do" (P9) and "direct what the final product should look like" (P8). But, she "[does]n't want to be the one actually coding" (P9). She wants to pass information "off to my coder" (P10) who will do most of the programming. She also believes that that coding knowledge "sets you apart" (P8) from other majors and helps "establish yourself as a very credible person" (P11) in the workplace.

As far as learning preferences, Connie is "more interested in knowing that I can understand what code is doing than writing it myself" (P9). She wishes there were "more questions where you just look at code and choose the right answer, instead of having to do it yourself all the time" (P10). She wants to learn in a way that helps her understand "overall what the process is like to accomplish certain things" (P9). She appreciates problems that "give you a basic structure to start with" (P11), like mixed-up code problems (aka Parsons problems [114]). When additional help is provided, like guiding comments in the code, Connie is "not as overwhelmed" (P8).

5.3.1.2 Alyssa the analyst

Alyssa wants a career like "data analyst" (P1) or "business analyst" (P7), where she can "use programming to solve problems that people are facing" (P6). She

Goal	Print the texts from all tags with class 'headline' from https://www.nytimes.com
Code	<pre> # Load libraries for web scraping import requests from bs4 import BeautifulSoup # Get a soup from a URL url = [] page = requests.get(url) soup = BeautifulSoup(page.text, 'html.parser') # Get all tags of a certain type from the soup all_[]_tags = soup.find_all() # Extract text from all tags texts = [] for []_tag in all_[]_tags: text = []_tag.text texts.append(text) # Do something with the texts [] </pre>

(a) High scaffolding (fill-in)

Goal	Print the texts from all tags with class 'headline' from https://www.nytimes.com
Code	<pre> # Load libraries for web scraping # Get a soup from a URL # Get all tags of a certain type from the soup # Extract text from all tags # Do something with the texts </pre>

(b) Some scaffolding (subgoals provided)

Goal	Print the texts from all tags with class 'headline' from https://www.nytimes.com
Code	

(c) No scaffolding (code from scratch)

Figure 5.2: Survey respondents were asked to rank and reflect on the usefulness of these code writing activities with different levels of purpose-oriented scaffolding. The directions for all activities were: "Complete the code that achieves the goal".

"[does]n't want a job where I have to heavily do programming" (P1), preferring "a mixture of programming and not really programming" (P6).

When learning, Alyssa thinks that "writing code, at least once you know all the basics, is really helpful, because then you can recall from memory" (P7). She feels that "being able to break it down and see what each individual line of code does is really helpful" (P2). She appreciates extra support "prior to starting writing a code just because I think it breaks it down much better and it helps me understand what the code is actually doing" (P2). However, after getting some practice, "writing your own code might be more helpful" (P6).

Alyssa took an introductory programming course from the computer science department, but in that course she felt she was "coding the project just to code the project" (P6) and "didn't understand how any of the coding [she] was learning applied to real life situations" (P2).

5.3.2 Survey results

I performed a survey of students in the same programming course to understand if findings from the focus groups generalized. Forty students responded

Table 5.1: Difference in attitudes about the highly scaffolded code writing problem (Figure 5.2a) between conversational programmers and non-conversational programmers. Attitudes were drawn from focus group quotes. Survey respondents were asked to rate their agreement on a 7-point Likert scale, Strongly Disagree - Strongly Agree.

Attitude	p-value	Conversational programmers who agree or strongly agree	Non-conversational programmers who agree or strongly agree
<i>This type of problem is less overwhelming because if I had to do it on my own I would have freaked out a little bit trying to remember what syntax to use, and what structure.</i>	0.560	62.5% (10/16)	55.6% (10/18)
<i>I feel like if code is always provided in problems like this, I will just end up forgetting to include things when I'm actually writing my own code.</i>	0.036*	33.3% (5/15)	72.2% (13/18)
<i>This type of question helps me understand how certain types of code should be structured.</i>	1.0	62.5% (10/16)	83.3% (15/18)
<i>I think if it's your first time looking at a problem like this, something with this structure would be more useful. But if you've been looking at it and working on it for a little bit, then writing your own code would be more helpful.</i>	0.046*	50.0% (8/16)	94.4% (17/18)
<i>I want to solve problems from scratch without being given any hint to what the final solution should be.</i>	0.103	18.8% (3/16)	38.9% (7/18)
<i>I prefer doing practice like this prior to writing a code because I think it breaks it down and it helps me understand what the code is actually doing.</i>	0.837	68.8% (11/16)	83.3% (15/18)

(29% of total course enrollment). Participants were offered a 1 in 5 chance to win a \$10 Amazon gift card.

Using responses about participants' future career goals and planned job responsibilities, I split respondents into conversational programmers and non-conversational programmers. Participants who planned a future career as a software developer, analyst, or who mentioned programming as a key job responsibility were not considered conversational programmers (n=19). Participants who planned a future career as a manager, designer, or who described another position (e.g. CEO) and did not list programming as a key responsibility were considered conversational programmers (n=18). Thus, I am comparing the comments between participants whom I classify as conversational programmers (like the Connie persona) and those whom I classify as non-conversational programmers (including analysts like the Alyssa persona and also computer science majors). Three respondents did not complete the questions about future goals and were not included.

Participants were asked to rank five programming learning activities (reading code, writing code, testing code, modifying code, and solving mixed-up code problems) by their helpfulness in preparing them for their future. Conversational programmers most often ranked code reading as the most helpful activity (8/16, 50%), while code writing was the activity non-conversational programmers most preferred (9/17, 53%). Non-conversational programmers overwhelmingly found mixed-up code problems (also known as Parsons problems [114]) to be the least helpful (15/17 ranked them last), while conversational programmers were more mixed, with less than half ranking them last (7/16).

Participants were asked to rank the usefulness of different types of code writing activities that provided different levels of support: writing code from scratch (no scaffolding), writing code with guiding comments (some scaffolding), and filling in parts of nearly completed code (high scaffolding) (see Figure 5.2). Both

groups ranked writing code with guiding comments (some scaffolding, Figure 5.2(b)) as the most helpful (11/16 of conversational programmers, 14/18 of non-conversational programmers). However, non-conversational programmers overwhelmingly ranked filling in mostly completed code (high scaffolding, Figure 5.2(c)) as the least helpful (16/18), while conversational programmers were more mixed about their preferences (4/16 ranked high scaffolding as the most helpful, 6/16 ranked it as the least helpful).

This difference in preference about scaffolding was also evident in participants' responses to Likert scale questions about their attitudes towards problems with this high level of scaffolding (see Table 5.1). I found that agreement with certain attitudes expressed by focus group members was significantly different between the two groups, according to a Mann-Whitney U test. While both groups agreed that a code writing problem with high scaffolding was less overwhelming and helpful preparation for writing code, non-conversational programmers were more likely to be concerned about their ability to write code on their own without additional supports.

5.3.3 Conclusions from the formative study

Conversational programmers and non-conversational programmers' differing goals inform the ways that they want to learn about code. Conversational programmers desire a basic and conceptual understanding of code, while non-conversational programmers want the ability to write code on their own. As a result, these two groups differ in the learning activities they value, and the amount of help they want while completing programming tasks. Conversational programmers prioritize code reading activities, and are welcoming of additional scaffolding that connects to code's purpose. Non-conversational programmers are more skeptical about heavy support while learning, as they value being able to write

code on their own.

These findings suggest that a highly scaffolded learning approach that prioritizes code's purpose may be strong match for the needs of conversational programmers. This approach could also potentially serve as a starting point for analysts and programmers, if an opportunity for fading of scaffolding is provided.

5.4 Defining Purpose-first programming

To meet the needs of a wider variety of novice programmers, I propose *purpose-first programming*, a new learning approach that emphasizes what code achieves while avoiding details of syntax and semantics. The core units of knowledge in purpose-first programming are *programming plans* [136, 134] drawn from the work of programmers in a particular domain. *Purpose-first scaffolds* help learners learn about and work with these plans so they can understand the purpose of authentic code and write new programs quickly. A purpose-first programming learning experience is driven by three design goals: it should be **brief**, it should prioritize code's **purpose**, and it should be **realistic** with respect to expert practice.

5.4.1 Identifying authentic, domain-specific programming plans

In prior plan identification work, plans were most often developed in the context of problems typical to an introductory programming course (e.g. [145, 134]). In purpose-first programming, plans are chosen to achieve domain-specific goals authentic to the work of programmers. In order to identify these plans, the corpus from which plans are drawn should reflect the workplace, not the classroom.

5.4.2 Expanding the definition of a programming plan to serve instructional needs

A programming plan is a commonly used code pattern, associated with the goal the code achieves [136, 134]. Programming plans are a powerful concept that integrates schema theory into the fields of computing education and program comprehension. However, in the literature, the definition of a “programming plan” is varied and sometimes vague.

I propose the following more precise definition of a programming plan to support instruction, assessment, and code writing during purpose-first programming.

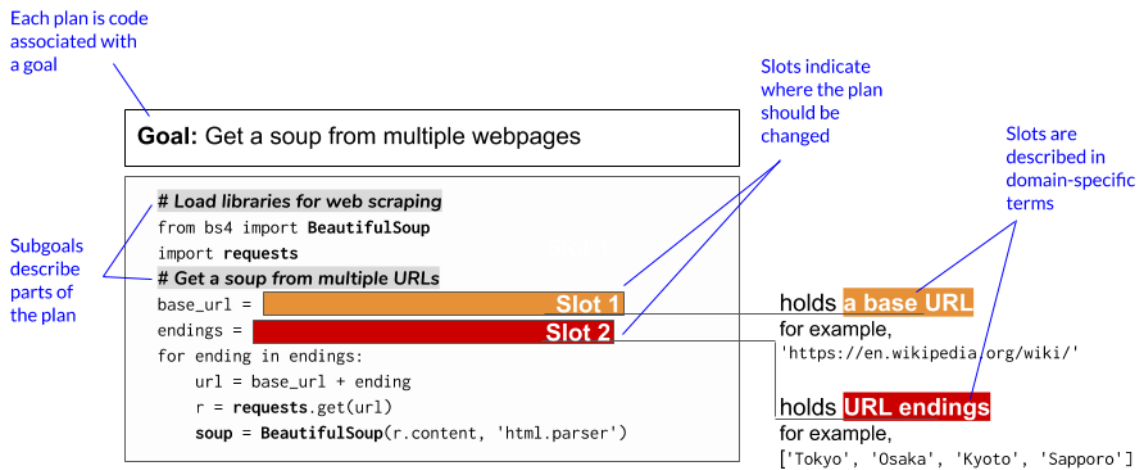


Figure 5.3: An example plan from the domain of web scraping. This plan achieves an authentic goal in its domain, and consists of multiple subgoals and slots. The slots define the space of relevant domain knowledge needed to work with this plan.

5.4.2.1 A plan is a frame with slots

The power of plans is that they can be re-used by programmers over and over again. The choice of what to modify and what to keep the same is crucial to applying a plan successfully. I define a plan as a frame [105, 128] that contains parts that can't be modified, and parts that can. The areas of a plan that can be changed are called “slots”. While prior plan editors allowed only numbers or strings to fill a

slot [64, 61], I will allow slots to contain not only literal values, but also other code. This allows plans to be nested.

5.4.2.2 A plan has subgoals as well as a goal

Since plans often contain many lines of code, additional *subgoals* can likely facilitate plan tracing, understanding, and integration. Subgoals describe what a small section of code achieves [99]. Research has shown that adding subgoals to code leads to better learning (improved retention and transfer) in less time than without the subgoal labels [99, 100]. Evidence suggests that subgoal labels support self-explanation behavior, in that the labels give students the language to use when explaining the programs to themselves [99, 107, 98]. Subgoal labels are also effective in helping students solve and learn from mixed-up code (aka Parsons) problems [106].

In purpose-first programming, each subgoal uses variables that were defined previously, and/or produces variables to be used in later subgoals. By tracing the input and output to each subgoal, learners can trace purpose-first code at a higher level of abstraction than evaluating each variable assignment and control structure [140]. Subgoals also suggest the way that plans can be combined: code from another plan can be added in a way that satisfies the subgoal label.

5.4.2.3 Slot contents are described with domain-specific concepts

Where a compiler sees only a string variable, humans understand that the variable represents a URL, a DNA sequence, or an address. Domain-specific concepts connect code to action in the real world. Similarly, domain knowledge is essential for understanding a plan's goal and how slots can be changed. In purpose-first programming, the content of slots is described in domain-specific terms. As a result, the types of objects that can go into slots provide a list of prerequisite knowl-

edge for using a plan.

5.4.3 Providing “glass-box” scaffolding to support learners as they work with plans

Shifting the focus of instruction from syntax and semantics to programming plans requires new interfaces for learning about code and completing programming tasks. In purpose-first programming, the ways that learners interact with code will be different than in standard IDEs. Instead of writing code by typing in text, purpose-first programming learners will *tailor* and *compose* plans. Instead of tracing changes in variables in memory, learners will trace with goals and sub-goals.

Experts have the ability to “chunk” [53] code into meaningful plan groupings, facilitating their reasoning about code [136, 120]. However, novice programmers don’t recognize or use programming plans as readily as experts [136]. In “purpose-first” programming tasks, learners will always work with authentic code patterns used by practitioners, which are likely to be complex. Learners will require *scaffolds* that support their ability to recognize plans, combine plans, and identify which parts of plans should be changed. A scaffold is an assistive mechanism that helps a learner complete a task they wouldn’t be able to do without added support [156, 162].

“Glass-box scaffolds”[68] are a model of support that can help purpose-first programming learners understand and write code easily while still having awareness of the full complexity of authentic code. A glass-box scaffold provides assistance to help learners achieve tasks, while not obscuring more complex, lower-level processes [68]. Learners can then focus on higher-level learning goals, while viewing information that can help with more complex tasks, if desired. By contrast, a “black-box” scaffold obscures details. Black-box scaffolds still facilitate

task completion, but block recognition of the full process needed for unassisted problem-solving.

In purpose-first programming, code should be “glass-boxed”. The text of a full program should be viewable by learners, as it demonstrates the authenticity of the learning activity and provides a potential on-ramp to future learning. At the same time, purpose-first scaffolds will direct attention towards key plan components, like slots, goals, and subgoals. This support will allow novices to complete realistic programming tasks while keeping cognitive load low [148].

5.5 Designing the purpose-first programming proof-of-concept curriculum

In order to evaluate the effectiveness of purpose-first programming, I developed a proof-of-concept curriculum for an audience of undergraduate novice programmers. This section describes the implementation of the proof-of-concept curriculum.

5.5.1 Building a set of plans

5.5.1.1 Choosing a domain

Web scraping, where a programmer extracts data from websites, is a common task for data scientists. During the formative study, I found that web scraping tasks also have disciplinary authenticity for students interested in conversational programming careers like user experience design and project management. These learners felt that the coverage of HTML and other web topics was relevant to their professional goals.

Web scraping requires the use of multiple feature-laden libraries (in my examples, I use the `BeautifulSoup` and `requests` Python libraries). Even basic

commands in these libraries involve complex mechanisms, including instantiation of objects, iteration on lists, and method calls that return custom object types. At the same time, web scraping programs tend to consist of a few similar strategies that are slightly modified to match each page’s particular layout. In practice, a small number of patterns are used over and over again, so a wide variety of web scraping tasks can be completed with only a few plans. The fact that web scraping is semantically complex but “planfully” simple makes it ideal for use in the proof-of-concept curriculum.

5.5.1.2 Identifying plans

I collected a corpus of BeautifulSoup web scraping files from a dataset of GitHub repositories collected in October 2019 [40]. The corpus included Python files containing the BeautifulSoup constructor and at least one instance of the BeautifulSoup method `find()` or `find_all()`. After removing duplicate files and files consisting of only unit tests, 100 files remained. I generated our initial set of plans after an analysis of the first 50 of these files.

I sought feedback on the authenticity of the plans in interviews with two experts who have used the BeautifulSoup web scraping library in their work. Both experts confirmed that all the plans were useful in web scraping, and that they have used the plans in their work. They also felt that useful web scraping tasks could be achieved using only the plans, although they described several tasks that would require use of additional plans, such as web crawling and scraping image files. The experts also provided suggestions for updates to deprecated libraries.

5.5.2 Creating activities

5.5.2.1 Choosing examples and tasks

Using only the plans I identified, I generated full programs that achieved an interesting web scraping goal on a real website. These programs achieved goals like getting all the cities from the location page of a pizza restaurant and getting the name of the “Pet of the Week” from a local humane society webpage. Two of these full programs were used as examples in the instructional part of the curriculum, and three were used in the activities in the latter part of the curriculum.

There were three types of activities: a writing task, where participants write a program that achieves a provided goal; a debugging task, where participants identify an error in a program and fix it; and an explanation task, where participants describe the purpose of an entire program in a sentence or two. The explanation program used the same protocol as “explain in plain English” problems [109], where learners can only read code, but not run it.

5.5.2.2 Sequencing activities

In the first part of the curriculum, learners complete instructional content about five new plans. This instruction is situated in examples. Participants learn and practice each plan after seeing it used within a full example program. Next, learners complete a series of activities that involve combining plans in ways not seen in the instructional content. The activities include a code writing, debugging, and explanation problem. The sequence of activities and plan usage across the curriculum is described in Figure 5.4.

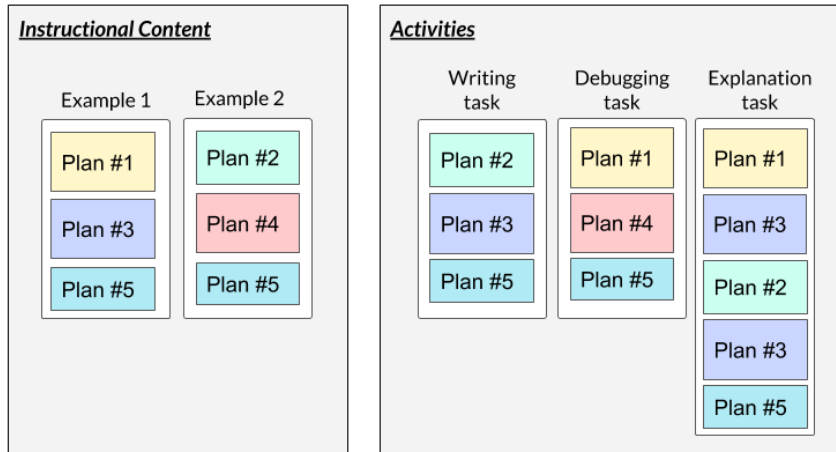


Figure 5.4: Plan usage across the curriculum. Activities use novel plan combinations not seen in instructional examples.

5.5.3 Selecting a platform

I developed the purpose-first programming proof-of-concept curriculum using Runestone, an open-source ebook platform with interactive feedback [45, 44]. Runestone is a popular platform for introductory programming learning that currently serves over 25,000 students a day. Instructors can create custom courses on Runestone that incorporate a wide variety of interactive components, such as runnable code and mixed-up code (Parsons [114]) problems.

5.5.4 Designing purpose-first support

5.5.4.1 Highlighting demarcates plans and slots

While existing highlighting features emphasize code's *syntactic structures*, I use highlighting to show code's *plan structures* (see Figure 5.5). Perceptual grouping of related symbols can improve performance in tasks like algebraic calculations [84]. This suggests that highlighting code that is part of the same plan and highlighting slots in a way that associates them with their natural language description in a subgoal may improve problem-solving.

Can you fix it? Here is the buggy code:

```
Plan 1: Get a soup from a URL
# Load libraries for web scraping
from bs4 import BeautifulSoup
import requests
# Get a soup from a URL
url = 'https://www.humanesociety.org/petsoftheweek/'
r = requests.get(url)
soup = BeautifulSoup(r.content, 'html.parser')

Plan 4: Get info from one tag
# Get first tag of a certain type from the soup
tag = soup.find('a', class_='pt-cv-none cvplbd')
# Get info from tag
info = tag.get('href')

Plan 5: Print the info
# Print the info
print(info)
```

Figure 5.5: Slot highlighting, plan goals, and subgoals assist learners as they debug this code.

5.5.4.2 Practice activities support learners in tailoring plans

Typical programming learning activities include writing code and predicting the result of code execution [155]. Purpose-first programming suggests new activities: identifying changeable parts of plans (see Figure 5.6a) and filling in plan slots to achieve a goal (See Figure 5.6b). These activities are quick to complete and minimize the opportunity for errors while focusing learners' understanding on key areas of code.

5.5.4.3 Examples and plan instruction are linked

To further emphasize code's purpose, all instruction in the proof-of-concept curriculum is situated in examples. Across two examples of complete programs, learners study all five plans. To make the connection between plans and their context of use more clear, students click on each plan in the example program to visit its instructional page and learn about the plan in detail (see Figure 5.7).

p5-3: Right now, this code gets the "text" from all 'h3' tags in the webpage. If you wanted to get the "links" from all the 'a', class = 'headline' tags in the webpage, which part(s) of the code below would you change?

```
# Get all tags of a certain type from the soup
tags = soup.find_all('h3')

# Collect info from the tags
collect_info = []
for tag in tags:
    # Get info from tag
    info = tag.text
    collect_info.append(info)
```

You are Correct!

p5-4: Fill in the plan in order to get the text from all <div class="headline"> tags on a webpage.

```
# Get all tags of a certain type from the soup
tags = soup.find_all('div', class='headline' )

# Collect info from the tags
collect_info = []
for tag in tags:
    # Get info from tag
    info = tag. text
    collect_info.append(info)
```

- Very close—but class should be class_!
- Correct.

(a) Learners *remember* parts of the plan that should be changed (b) Learners *apply* their knowledge to complete a plan

Figure 5.6: Practice activities contain subgoal label scaffolding, and focus only on knowledge about plan slots.

5.5.4.4 Staged code writing supports learners in assembling and tailoring plans

In a traditional editor, learners type code character by character. In purpose-first programming, learners make use of plan structures to assemble and tailor plans.

Drawing inspiration from mixed-up code (aka Parsons) problems, where learners arrange lines of code into the correct order, I developed a code writing activity in three parts (see Figure 5.8). First, learners pick from a bank of plan goals and drag selected goals into the correct order. Next, learners repeat this task, but with plan code instead of goals. Finally, learners fill in the slots in the code they have assembled.

5.5.5 How this prototype meets the design goals

5.5.5.1 Purpose-first programming is brief

Our prototype curriculum offers a *brief* learning experience where students can learn and apply the basics of web scraping in about an hour. This short timeline is possible because no content is taught unless it helps learners understand and modify one of a small number of plans. For example, the official BeautifulSoup

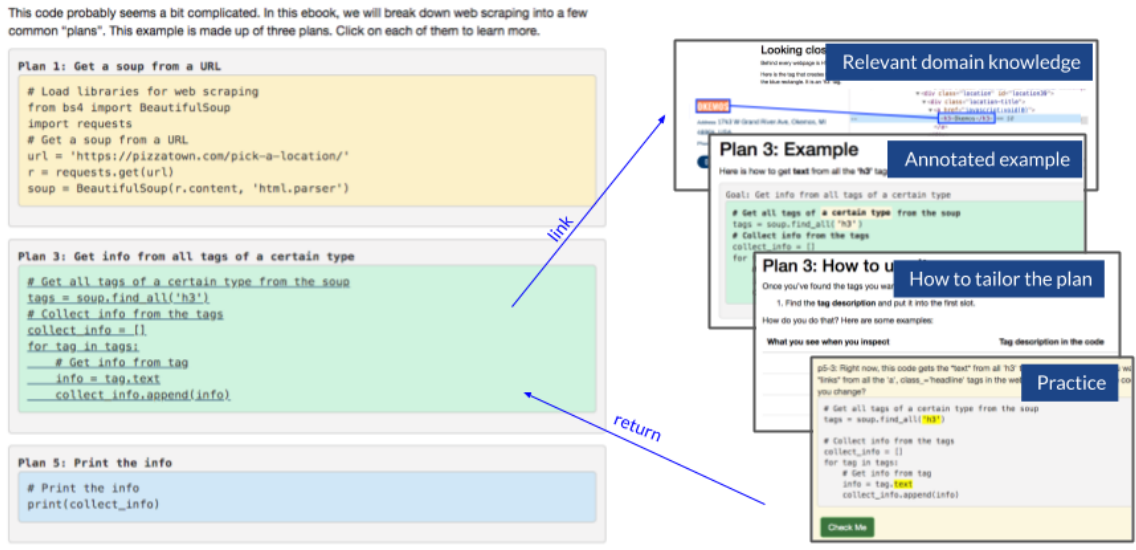


Figure 5.7: All plan instruction is situated in examples. Learners first view and run a complete program example, then learn about how each plan contributes to the full program.

documentation begins by explaining that a `soup` object is a nested data structure [1], but in our purpose-first programming curriculum it isn't necessary to explain what a `soup` object is. That level of detail isn't necessary to assemble and tailor plans.

Our proof-of-concept curriculum is also brief because relevant information is available *just-in-time*, minimizing the search for information. For example, relevant domain knowledge is covered just before it is needed to complete a plan. Links to plan information are available when solving problems, so key examples are readily accessible.

5.5.5.2 Purpose-first programming prioritizes purpose

Conversational programmers value "high-level", conceptual, and application-oriented information about code [28, 29, 157]. This purpose-first programming curriculum makes code's purpose clear at three different levels.

At the level of entire programs, learners only see examples that achieve a meaningful web scraping goal. Examples are not toy problems, but programs that col-

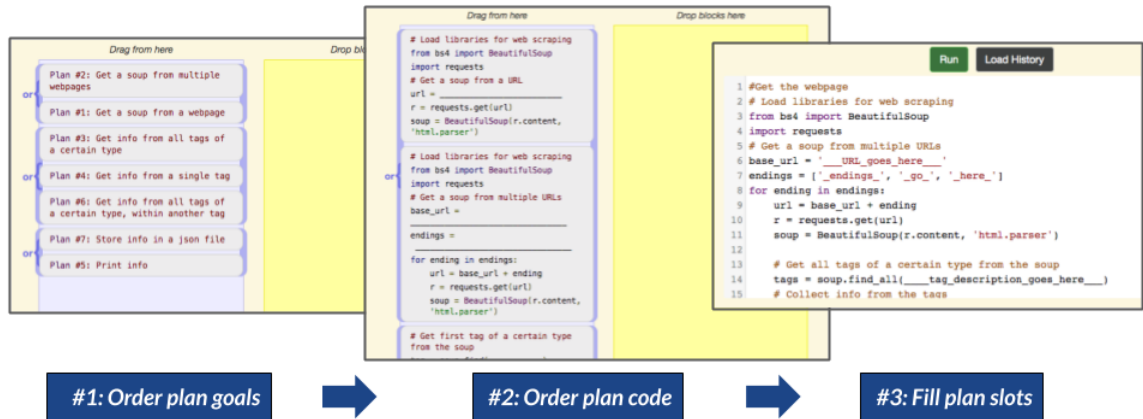


Figure 5.8: Writing code takes place in stages. Learners first assemble plans, and then fill in plan slots.

lect data that could potentially be used by data scientists. Learners can see the purpose of the programs they are learning from. The examples are also runnable, so learners can be assured that programs truly work as intended.

At the level of individual plans, each plan is introduced in the context of a full example program, showing its relevance to meaningful code. By grounding plan instruction in these examples (see Figure 5.7, learners can see how that plan’s goal contributes to the program’s goal, and understand the purpose the plan serves in the example.

At the level of the code within plans, annotation with subgoals creates a natural language layer on top of code that explicitly describes code’s purpose. With these supports, connections between the code and what it achieves may become more clear. Code is available to view, but it can be ignored in favor of these descriptions of code’s purpose.

5.5.5.3 Purpose-first programming is realistic to the work of programmers

Three features communicated the disciplinary authenticity [131] of the content in the proof-of-concept curriculum.

First, learners were informed at the beginning of the curriculum that the cur-

ricular content was designed based on an analysis of Github files and expert interviews.

Second, activities demonstrated how to get information from real and recognizable websites. Examples and activities incorporated code that scraped the website of a well-known local pizza restaurant, RateMyProfessors.com, the website of the local humane society, and faculty homepages. This authenticity was reinforced with images and GIFs that showed exactly how to obtain necessary information from these websites' HTML (although participants never had to do this process themselves).

Third, the code of complete programs was often available and often runnable. As a part of the situating examples and activities, participants could view the full program code. The code was not hidden, even though participants could focus on goal and subgoal labels and ignore code if they wished. Participants could compile and run the code to see that the results were truly returned from the websites.

5.6 Method

To evaluate the purpose-first programming proof-of-concept curriculum, I performed an lab study with novice programmers (see Figure 5.9 for an overview of the study design).

5.6.1 Study Design

First, participants completed instructional content, where they learned five new plans. A researcher guided the participant through the instructional content, reading instructional text aloud and answering questions about the exercises as requested. Next, participants completed code writing, debugging, and explanation activities that combined the plans in ways that were not seen in the instructional

content. Participants were asked to perform a concurrent thinkaloud while completing these tasks. Finally, participants completed a semi-structured interview to understand their motivation for learning with purpose-first methods.

The study was conducted over video call, which was recorded and transcribed. Each session lasted 90 minutes in total. Each participant received a \$50 Amazon gift card. The study took place in the week before the fall semester began.

5.6.2 Recruitment and participants

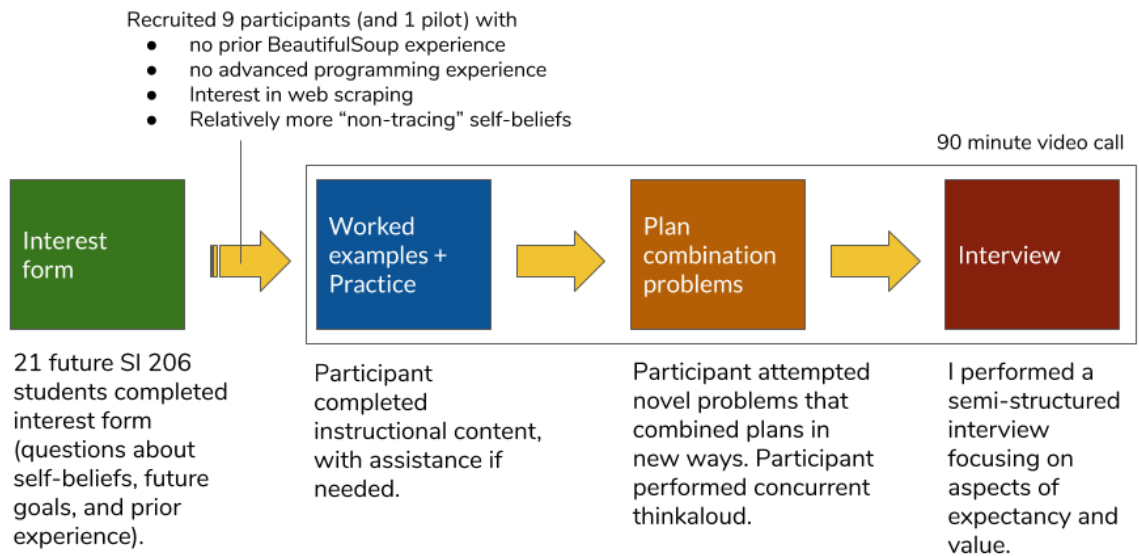


Figure 5.9: Overview of the study design

5.6.2.1 Recruitment criteria

I recruited undergraduates enrolled in a data-oriented programming course (the same course studied in the formative study) at the University of Michigan. Since the study took place before the start of the semester, participants had not yet seen any of the content from the course.

After sending an interest survey to students registered in the course, I recruited participants with characteristics that suggested they would be motivated by pur-

pose-first programming and didn't already know the content of the curriculum. I recruited participants who (a) had no prior experience with the BeautifulSoup web scraping library, (b) were interested in web scraping, (c) had not taken advanced programming courses in the computer science major, (d) were interested in non-software engineering careers, and (e) responded below average in either self-efficacy for code tracing activities or value for code tracing activities. Thirty-one students completed the interest survey, sixteen were recruited, and ten participated in the study (one pilot and nine full participants).

5.6.2.2 Participants and their goals

Nine students participated in the study. All participants were female. Six participants were majoring in Information in the User Experience track, two were majoring in Business, and one was majoring in Information in the Information Analysis path. Eight participants had taken one prior programming course in college (either an introductory data-oriented Python course in the information major (6), an introductory C++ course in the Computer Science major (1), or an introductory MATLAB and C++ course for engineers (1)). One participant had taken both the data-oriented Python course and the C++ course in the Computer Science major.

Although participants had similar levels of prior coursework, they had a variety of goals. Five participants indicated that they didn't want to spend much time programming in their careers, which included user experience designer, product manager, and/or product designer. These five participants can be categorized as "conversational programmers" [28, 29]. Participants were recruited based on survey responses that indicated they were targeting non-software developer positions, however, during the interview portion of the study four participants expressed a desire to spend significant time programming in their career. Two participants were interested in careers often associated with conversational program-

mers — product management and UX design — but indicated that they wanted to spend significant time doing front-end programming. Another participant expressed a desire to become a data analyst, and another a software developer.

5.7 Evaluation of learners' problem-solving

After completing the instructional content (average time 31 minutes), learners attempted scaffolded code writing, debugging, and explanation activities while performing a concurrent thinkaloud. These activities involved different plan combinations than learners had seen in content up to that point (see Figure 5.4). Participants were allowed to reference plan instructional content while solving these problems, and received some feedback after each attempt. If a participant gave up on a problem or was floundering, the researcher provided a hint.

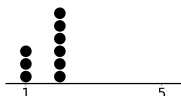
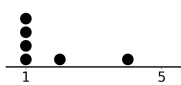
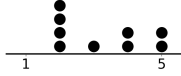
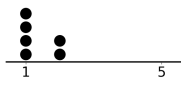
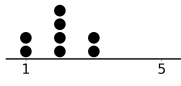
To understand *if* novice programmers are able to write, debug, and explain web scraping programs using the scaffolds provided by the curriculum, I quantified markers of their success, including rate of completion and assistance used. To understand *how* novice programmers were able to write, debug, and explain these programs, I analyzed participants' application of plan knowledge and use of the scaffolding provided.

5.7.1 Learners were able to complete scaffolded writing, debugging, and code explanation tasks

Participants were largely able to apply the plan knowledge they learned in the instructional content to complete the various scaffolded programming activities (see Table 5.2). The majority were able to complete the tasks without intervention by the researcher, although problem-solving often took multiple attempts.

Success varied by activity. Participants completed plan ordering tasks quickly

Table 5.2: Participants' success and time to completion on scaffolded activities (n=9).

Activity	Succeeded without researcher hint	Mean time to completion (min)	Frequency distribution of number of attempts
Writing 1 (order plan goals)	100%	1:31 (sd=0:40)	
Writing 2 (order plan code)	100%	2:00 (sd=1:09)	
Writing 3 (fill plan slots)	67%	7:58 (sd=2:13)	
Debugging	89%	2:19 (sd=1:36)	
Explanation	56 %	6:07 (sd=2:49)	

and without researcher assistance, even though several participants needed a second attempt to solve these problems. Filling in plan slots and explaining a full program were the most challenging for participants, and also required the most time. The code explanation problem was the most challenging of all, possibly because it involved the most distant transfer. In the explanation task, participants summarized the functionality of a program consisting of *five* plans, while the other activities and instructional examples had contained three plans (see Figure 5.4). This problem involves farther transfer than the other activities due to its structural differences.

5.7.2 Participants used purpose-first scaffolds to apply plan knowledge and complete tasks

Were the purpose-first scaffolds a reason for participants' success in the activities? I observed that participants did make use of key features of purpose-first

programming as they problem-solved, demonstrating their ability to apply plan knowledge. In this section, I describe how participants used goal and subgoal information and plan reference material to complete the activities.

5.7.2.1 Participants used goal and subgoal labels to complete tasks, some tasks more than others

One type of purpose-first scaffolding in the curriculum was the inclusion of purpose-oriented goal and subgoal labels to facilitate problem-solving with code (see Figures 5.5,5.6,5.7,5.8). To understand whether participants referenced these scaffolds during completion of the writing, debugging, and explanation activities, I counted mentions of goals and subgoals during participants' thinkalouds.

Coding process Two coders coded two participants' transcripts (22% of total data¹) for goal and subgoal mentions, and reached 92% agreement². One coder then completed identification of goal and subgoal mentions for the remaining participants.

Results Participants often mentioned goals and subgoals while solving all problem types (see Table 5.3). Mentions were more frequent during some activities than others. Mentions of goals and subgoals were most common when participants ordered plan goals, ordered plan code, and explained code.

On the Writing 1 (order plan goals) activity, the average number of mentions of goals and subgoals was highest. One potential reason is that plan goals were the main feature of this activity (see Figure 5.8), and plan code and subgoals were

¹Within best practice of 10%-25% [161]

²I use simple agreement as the metric to measure reliability. Mentions of subgoals are relatively less common among the unstructured thinkaloud narrative, and so reliability measures like Cohen's kappa and Krippendorff's alpha would be inflated by the large amount of agreement on units not labeled as subgoal labels. Simple agreement is a more conservative measure of agreement for inter-rater reliability in this case [82]

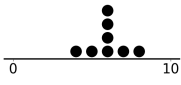
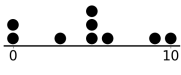

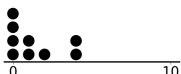
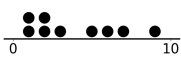
not present. The relatively greater mentions can be explained by the fact that participants had little else to talk about during their thinkalouds on this problem. In all other activities, however, plan code and associated subgoals were visible during problem-solving. Despite the similar amount of goal and subgoal information, mentions of subgoals varied across the final four problems. When filling plan slots in Writing 3, participants had a relatively low number of goal and subgoal mentions, even though this activity had the longest completion time on average. Participants were also less likely to mention goals and subgoals in the debugging activity. By contrast, when participants ordered plan code in Writing 2 and explained the purpose of a program, goal and subgoal mentions were more frequent.

These differences can be explained by the different types of plan knowledge required to complete each activity. When the focus of the activity was on *modifying plan slots*, as in Writing 3 (fill plan slots) and Debugging, participants did not mention goals and subgoals as often. By contrast, when the focus of the activity was *choosing relevant plans*, as in Writing 2 (order plan code), or *understanding how a plan contributes to an entire program*, as in the Explanation activity, participants were more likely to use subgoals in their reasoning. It appears that participants saw subgoal knowledge as more useful when understanding how different plans work together to achieve a larger goal, but less useful when they were focusing on how to fill plan slots to complete code or find errors.

5.7.2.2 Some participants used subgoals to reason about code, while others focused on code's identifiers and control flow

On the same activities, participants varied widely in their mentions of goals and subgoals (see the distributions in Table 5.3). Individual verbosity is certainly a factor [27]. However, I identified another difference at play: on problems where both code and subgoals were available, some participants focused more on sub-

Table 5.3: Participants' mentions of goals and subgoals during their thinkaloud on each activity.

Activity	Mean number of mentions	Frequency distribution of number of mentions
Writing 1 (order plan goals)	6.0 (sd=1.2)	
Writing 2 (order plan code)	4.8 (sd=3.5)	
Writing 3 (fill plan slots)	1.6 (sd=1.4)	
Debugging	1.3 (sd=1.7)	
Explanation	4.0 (sd=2.9)	

goals during their thinkaloud, while others focused more on code.

Even at the same stage of problem-solving, some participants mentioned subgoals as they described their process of understanding or choosing a plan, while others mentioned variables, functions, loop structures, and other features of code (see Table 5.4). For example, different approaches were employed while selecting and ordering plans to build a full program in Writing 2. In her thinkaloud, P2 describes the plans she is choosing between in terms of each plan's first subgoal: *"Choose between these three. **Get first tag of certain type from soup, get all tags of a certain type, get first tag of a certain type**"* (mentions of subgoals are bolded). P2 also describes her choice in the language of the subgoal label, saying *"I think it's this one, we want **all tags**."* By contrast, P5 doesn't mention subgoals at all during her thinkaloud on this problem. She begins to evaluate this part of the task by reading part of the code, saying *"now I do tags of soup dot find"* (mentions of code are underlined). P5 makes her choice between those same plans based on the recognition of functions from earlier examples. She says, *"the code is the same except for that this second one has find_all in it as well and I don't know if we need that."*

Subgoal labels were a part of problem-solving for many participants, providing a vocabulary for reasoning about code using natural language. However, subgoals were not the only aspect of the task that learners could use to make sense of code. Some participants followed execution flow to understand programs, or focused on code features as "beacons" to identify and understand plans [50, 121].

5.7.2.3 Participants were able to identify and use relevant plan reference pages for help

When filling in plan slots in the final stage of the writing activity (see Fig 5.8), participants frequently returned to plan information pages for help. Participants moved back and forth between code editing, reading of instructions, and information about plans as they completed their web scraping program (see Figure 5.10 for a trace of all participants' actions while solving this problem). Referencing plan instructional content, such as directions about how to use each plan and examples, allowed participants to fix bugs in their code. Participants made an average of 4.0 visits to plan instructional pages ($sd = 2.6$) during this activity. The number of visits to plan pages varied widely between participants, from as few as one visit to as many as eight. As a group, conversational programmers visited about twice as many plan pages as those who planned to become programmers (see Table 5.5).

Participants demonstrated that they could identify and make use of relevant plan information for their problem-solving. Participants typically referenced the plan page most relevant to their current error or code authoring activity. Seventy-eight percent of all visits to a plan reference page were for a plan where the participant's code contained an error or was incomplete at the time of the page visit. Note that this definition of "most relevant" penalizes visits to plan pages where participants are "double-checking" their work, and therefore serves as a conservative estimate of relevance. The ability to identify relevant plan pages and then

Table 5.4: Examples of different participants focusing on subgoals or on code at the same step in an activity. Mentions of subgoals are bolded, and mentions of code are underlined. All quotes are from conversational programmers.

	Focus on subgoals	Focus on code
Writing 2 (order plan code), choosing between Plan 3, Plan 4, and a distractor plan	<i>Okay. Um... First we want to start with the getting tags from the soup. Choose between these three. Get first tag of certain type from soup, get all tags of a certain type, get first tag of a certain type. I think it's this one, we want all tags. Okay. (P2)</i>	<i>Okay, I don't, I think this is gonna be later and then now I do <u>tags of soup dot find</u>. <u>Find versus find_all</u> I know. We learned that. I feel like... I'm not sure which one. So I'm just gonna do that one because I don't know if the code that's, because the code is the same except for that this second one has <u>find_all</u> in it as well and I don't know if we need that so I'm just not going to do that for now. (P5)</i>
Explanation, understanding Plan 4	<i>And then tags of a certain type from the soup. So this is the exact place it wants to get it from [hovers over plan slot with her mouse] , which would be here [mouses over the corresponding part of the HTML]. So it's basically saying that it's all this, the link and then the title, so what it actually leads you to. And then <u>collect info</u> [unclear if she means subgoal label or code]. This is because it's getting it from a certain type, so it has to go into something else and then they want this specific URL. I'm guessing that's what they need for that type. (P6)</i>	<i>Oh, okay. Okay. So I guess it's gathering all of those links in the get('href') and it's putting it into <u>this dictionary</u>. I think that's a dictionary. Don't remember Python super well. And then from the multiple web-pages... I don't know what the collect... Oh, oh, just kidding. So then it's just going through the SI website and adding those URLs we got and put into the <u>collect_info dictionary</u>, I think. <u>And then it's reading them in the soup</u>, I think. (P1)</i>

apply knowledge to edit code correctly shows that participants are able to work effectively with the plan groupings prescribed by the curriculum.

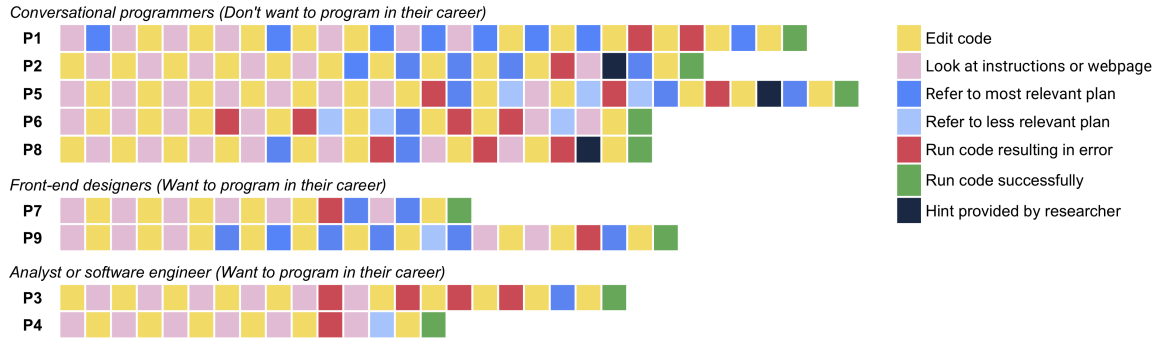


Figure 5.10: Traces of participants’ activities when solving code Writing part 3 (fill plan slots)

Table 5.5: Participants’ visits to plan reference pages in Writing Activity 3. “Most relevant” plans are plans where the participant’s code contained an error or was incomplete at the time of the page visit.

	Average # of most relevant plans visited	Average # of less relevant plans visited	Average total # of plans visited
All participants (n=9)	3.1	0.9	4.0
Conversational programmers (n=5)	3.8	1.2	5.0
Non-conversational programmers (n=4)	2.3	0.5	2.8

5.7.2.4 During code writing, participants could identify information about how to fill plan slots

Participants visited relevant plan information pages while tailoring plans, but how exactly did they make use of plan information? I analyzed two participant’s thinkalouds during the Writing 3 (fill plan slots) activity to investigate the information that learners searched for.

P9, a Business major interested in program management or user experience design, visited plan pages seven times, and was able to complete the Writing 3 activity without researcher assistance. Examination of her actions and her thinkaloud show that she was able to glean information about how to fill plan slots from the instructional content, and describe the reasoning for filling plan slots in her own

words (see Figure 5.11).

P9 used plan reference pages to answer two main questions: *What goes in a slot?* and *How is it formatted?* P9 spoke in short definitions or productions to describe what content should fill slots, e.g. *"that's the part of the URL that doesn't really change"*, and *"this would be `text` because I'm trying to find a comment"*. P9 corrected an error in her code by applying one of these productions: she determined that because she is working with multiple tags, she should fill the last slot with `collect_info` rather than `info`. Interestingly, P9 never mentioned syntactic terms like "string", "parameter", "argument", or "function" at any point in this thinkaloud. Instead, her talk was focused on aspects of web scraping, like "URL", "comment", "link", and "tags".

Understanding details of formatting was also important for P9 to complete the code. Although she filled in all slots with the correct information, in one case she had made a formatting error (omitting an underscore). After identifying the location of the error because of interpreter feedback, she visited the most relevant plan page and was able to correct her mistake.

5.7.2.5 Failure to find relevant information occurred after shallow understanding of or when examples were too dissimilar from an attempt

P5, an Information major interested in user experience design, was less successful in her search for relevant information. She was able to fill many slots correctly, using propositional statements and definitions as P9 did. However, she could not determine how to fix an error in one particular slot. After expressing frustration that she was unsure how to continue, the researcher provided a hint and she was able to fix the error.

P5 is the only participant that referenced a plan page for a plan that was *not* a part of the code in the activity. The circumstances around her multiple visits to this

plan page provide two reasons why novices may struggle to identify relevant plan knowledge. First, they might perform a shallow matching of subgoals to goals, and second, their current attempt may have too much structural dissimilarity with the correct plan structure.

As P5 attempted to fill in a plan slot she was unsure about (*"I honestly forget what I'm supposed to put here"*), she tried to find a useful example by reasoning from plan subgoals and goals. She mentioned in her thinkaloud that she believed the subgoals in the code would direct her to a relevant plan page, saying *"There's these comments left, to direct me to those [plan pages], that makes sense."* She read the subgoal closest to the slot that she was trying to fill ("get info from tag"), found a plan goal that used similar wording ("get info from a single tag"), and visited the plan page with that goal to look for relevant information. While in this case the subgoal and plan goal were similar, that plan was not the most relevant to her current task. However, in the curriculum, plans had some redundancy—sometimes the same lines of code appeared in multiple plans. P5 was able to find an example that was similar enough to the plan she was working on completing, and she was able to complete the slot successfully.

P5 visited this less relevant plan page again for a different reason. P5 had filled a plan slot with an incorrect value and was looking for relevant plan information to correct her error. Specifically, P5 had typed the tag description *twice* (`' div' , class_=' Comments__StyledComments-dzzyvm-0 dvnRbr' , ' div' , class_=' Comments__StyledComments-dzzyvm-0 dvnRbr'`), instead of only *once* as she was supposed to. P5's thinkaloud (see Figure 5.12) shows her repeated, unsuccessful search for an example with two tag descriptions rather than only one.

P5 searched across two different plan pages as she looked for a relevant example. As she looked at the most relevant plan information page, P5 said, *"I want to make sure that because there's two, a comma is what's needed to separate those. But I'm not*

seeing an example of that so don't know how that works." After not finding anything on the most relevant page, she searched the less relevant plan page, with the same result ("Is there not an example of how to do it with two?"). P5's attempt to fill a plan slot was so structurally dissimilar to the plan examples she viewed that she believed none of the plan information she viewed was relevant.

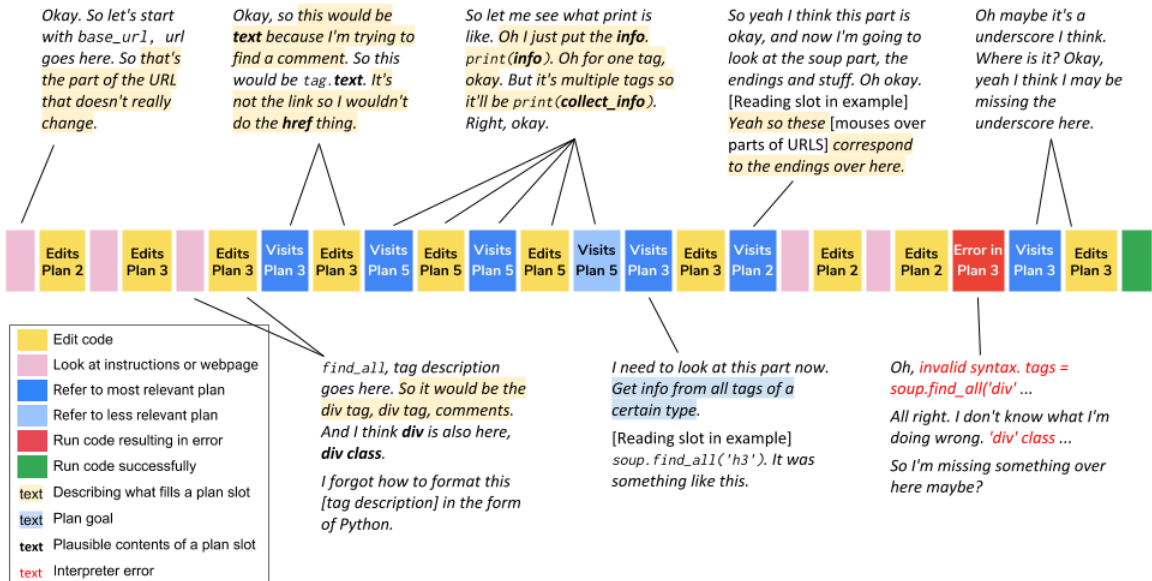


Figure 5.11: Selected quotes from P9's thinkaloud mapped onto the actions she took while solving Writing Activity 3 (fill plan slots).

5.7.3 Discussion

With the support provided by the purpose-first programming curriculum, novice programmers with a variety of future goals progressed from no prior experience with the BeautifulSoup library to being able to complete scaffolded coding activities after about half an hour of instruction. This timeline is greatly accelerated over the typical pattern of instruction and coding activities for BeautifulSoup content in the data programming course, where instruction spans approximately five hours of lecture and discussion section time, and assigned coding activities are expected to require at least 30 minutes each [personal communication with in-

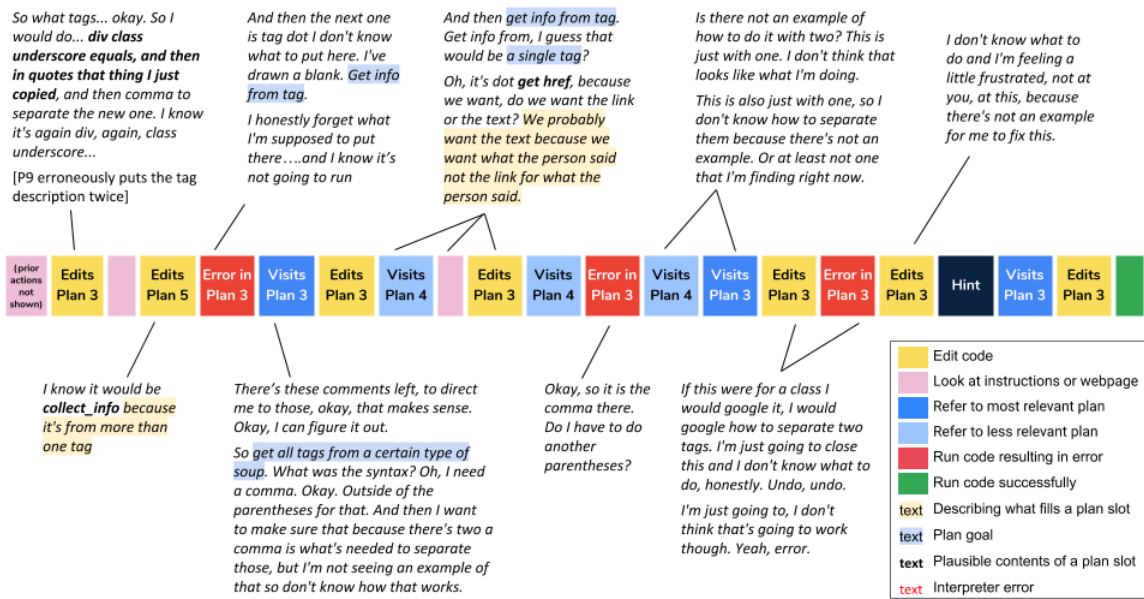


Figure 5.12: Selected quotes from P5's thinkaloud mapped onto the actions she took while solving Writing Activity 3 (fill plan slots).

structor, November 2020]. The proof-of-concept curriculum achieved the design goal of being *brief*.

Through analysis of the thinkalouds and the actions participants took while solving problems, it's clear that purpose-first scaffolds assisted learners as they built, edited, and understood programs. Participants utilized goal and subgoal labels, particularly as they made choices between plans and assembled or understood whole programs. While editing a program, (most) participants were able to identify which plan was most relevant to their current problem-solving. These findings provide encouraging initial results that novice programmers can make effective use of purpose-first support in a purpose-first programming curriculum.

Participants did differ in their use of the scaffolding. Some participants were more likely to reason about programs using subgoals, while others reasoned using code. As a result of the "glass-boxed" approach of purpose-first programming, participants have the flexibility to focus on either the code (as in a typical programming activity) or the purpose-oriented subgoals (unique to purpose-first

programming). This feature suggests that purpose-first programming can accommodate both learners who do not yet feel comfortable reasoning about code, as well as those who will later transition to deeper learning about code.

5.8 Evaluation of learners' motivation

Does purpose-first programming motivate novices who aren't well-served by existing instructional approaches? If so, how exactly does this curricular approach contribute to a feeling of motivation for future learning? To answer these questions I interviewed participants after they completed the purpose-first programming curriculum, focusing on participants' feelings of motivation, their sense of success, their value for the tasks they had completed, and their future goals and self-beliefs.

5.8.1 Analysis approach

Interviews were screen- and audio-recorded and transcribed. I performed deductive reflective thematic analysis [16] on the interview transcripts to conceptualize themes and situate them within the Eccles Expectancy-Value Model of Achievement Choice [42]. This analysis was *deductive* because I analyzed the data with a framework in mind. This analysis was *reflective* because I took an active role in developing a coding scheme, with the understanding that I interpret data through my own lens of prior experience and understanding [16].

I used values coding during the coding stage of thematic analysis to highlight the ways participants described their identity and motivations [117, 126]. Values coding focuses on the belief system of the participants by identifying values (the importance of people, things or ideas), attitudes (the way participants think and feel about people, things, or ideas), and beliefs (rules and interpretations) in their

talk. After I conceptualized themes using the thematic analysis process outlined by Braun and Clarke [17], I mapped those themes onto the Eccles Expectancy Value Model of Activity Choice [42] in order to understand the process of motivation for these novices.

5.8.2 Participants were motivated to learn with purpose-first programming in the future

When asked if they would want to continue learning with plans in the future, all nine participants said that they would. P1, an aspiring product manager, said, *"I could see myself really liking this curriculum for other topics."* *"If you had any other thing besides web scraping I'd gladly sign up,"* said P9, a student interested in user interface design and management. P7, who is interested in becoming a front-end developer, said, *"If I could have one of these to do every week, and then in two years I would be so much more well-versed in Python, I would absolutely do it."* Both conversational programmers and aspiring professional programmers were motivated to learn with plans in the future.

5.8.3 Learners perceived the purpose-first programming curriculum as having low cognitive load

Participants consistently described learning with plans as "easy", "clear", and "helpful". In comparison to other instruction, participants often described the curriculum as providing more assistance to help them learn and complete tasks.

"Previously it was never building up to writing the code on my own. It was either really, really easy questions, or really, really hard questions, and there was no transition that really made it stick in my brain." - P1

According to participants, purpose-first programming decreases cognitive load

in two ways: limiting the amount of information learners need to focus on, and assisting in the application of information.

5.8.3.1 Plans allow learners to focus on less

Participants described plan-based activities as limiting attention to smaller and more manageable parts of code. Learners frequently used language like “breaks down” and “breaks up” when describing purpose-first programming. P3, a student interested in data analysis, said that the curriculum *“breaks up a bigger goal into smaller goals, that are more easy to understand, instead of just wrapping your head around the whole thing at once.”* P7, an aspiring front-end developer, agreed, saying that when learning with the curriculum she could *“Just think about this stuff, and then think about this stuff.”* For her, plans involved *“One at a time instead of looking at it all.”*

When creating code, participants felt that choices were constrained to a manageable number. P5, a student interested in UX design, described this phenomenon by saying, *“I think its easy just because it’s one or the other. Is it just one tag, multiple tags, is it one site, is it multiple sites, is it text, is it a link kind of thing.”* P8, a future designer, concurred, describing how limited choices in the curriculum made a seemingly complex topic straightforward:

“The concept of web scraping, at first is kind of intimidating, because it’s like, how do you get all this information? But the way that the plans are written out, do you have one URL or do you have two? Are you trying to get links or are you trying to get text? I feel like the way it was explained was very clear.”

5.8.3.2 Plans show how to apply knowledge

Participants recognized plans’ connection between code and its application. P8 described plans in terms of their goals, calling them a *“very clear group, like, this is*

one that prints your file information. This is what sets up the URL." Participants felt that this association made problem-solving simpler. *"Plans make it easier to choose what method you should be using for each type of situation,"* P2 explained. P4 defined plans as *"sort of like a formula in a sense, like you always know when you're trying to do this particular thing. Follow this, use this code snippet."* For these learners, plans made it clear what to do when they had a particular goal in mind.

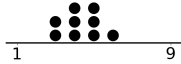
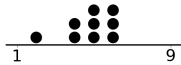
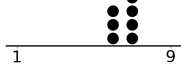
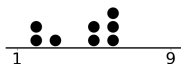
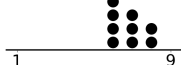
Understanding how to decompose tasks was a challenge participants described in their typical programming learning. P6 recounted that in her previous programming course she struggled with *"having a very general destination and not knowing how to get there. So I thought that [the curriculum] was really helpful."* Participants frequently described plans as *"steps"*, one of a series of actions towards a goal. These steps assisted with code creation and understanding. P7 described a contrast with other programming activities: *"Usually I'm just used to being like, 'Okay, here's point A and B, and I want to get here. How do I get here?' Instead of A, B, then we get to C, then we get to D, and that's the final thing."* In the purpose-first programming curriculum, the difficult process of identifying sub-goals was removed, resulting in an easier problem-solving experience.

5.8.3.3 Self-reported cognitive load ranged from moderately low to moderately high

Participants rated their cognitive load after each writing, debugging, and explanation tasks in the curriculum, using the scale proposed by Paas [111]. Writing 3 (fill plan slots) and Explanation had higher cognitive load ratings than the other tasks (see Table 5.6). These tasks also had longer average time to completion and lower success rates without assistance (see Table 5.2).

Participants most often described their mental effort in the middle values of the scale (4-7). For the more challenging Writing 3 and Explanation activities, partic-

Table 5.6: Self-reported cognitive load on scaffolded activities (n=9). Cognitive load is measured on a 9-point scale (Very very low mental effort (1) - Very very high mental effort (9) [111]).

Activity	Mean cognitive load	Frequency distribution of cognitive load
Writing 1 (order plan goals)	4.3	
Writing 2 (order plan code)	4.8	
Writing 3 (fill plan slots)	6.7	
Debugging	4.4	
Explanation	6.8	

Participants most often described their mental effort as “Rather high mental effort” or “High mental effort”. For the less challenging Writing 1, Writing 2, and Debugging activities, participants most often chose scale values of “Rather low mental effort”, “Neither low nor high mental effort”, and “Rather high mental effort”.

5.8.3.4 Theme summary

Participants described the activities as being easier and requiring less effort than standard programming activities. Cognitive load ratings show that participants rated activities as having moderate cognitive load. Participants described the lower cognitive load of purpose-first programming as a result of needing to focus on less information and having clarity about how to apply the knowledge they’d learned.

5.8.4 Participants felt success and enjoyment, which came from understanding and completing problems

When asked if they felt successful and if they enjoyed the learning experience, participants largely agreed. Success and enjoyment were interrelated: being able to understand and complete problems led to both success and enjoyment.

5.8.4.1 Participants largely felt successful, despite struggles and resource use

Participants often described a feeling of success after completing activities in the curriculum, even though they didn't necessarily expect it. *"I honestly feel like I did better than I expected to do,"* said P7, speaking generally about the experience. When asked if she felt successful at web scraping, P8 replied, *"Yeah, which, like I said, I wasn't really expecting, so that was good."*

These learners felt successful, even though they often had to use resources and hints. All but one participant described themselves as successful on the code writing problem, even though all participants used plan reference page resources, and 33% received a hint from the researcher. *"Even though I had to look back at previous examples I still figured it out in the end, so I feel good,"* said P9. When describing her feeling of success on the code writing problem, P1 said, *"I did mess up the tags text with the info section. Because I didn't realize that I had the info one. But I think at the end it worked."* P1 recalled struggling, but still felt successful overall.

5.8.4.2 Feeling that they understood and completed activities was associated with participants' success

These learners' talk about success was often paired with a statement about accurately completing problems in the curriculum or having a sense of understanding the content. For instance, when P7 talked about feeling successful after completing the curriculum, she said, *"I feel like I was able to comprehend and understand*

the questions and complete them."

Participants related their perception of low cognitive load to their ability to understand and complete problems. Reflecting on the entire curriculum, P8 said, *"it walked me through so clearly that I was able to complete the tasks very easily and feel like I actually did it successfully."* Completion with low effort led to a feeling of success.

P5, a conversational programmer, described her confidence building throughout the curriculum.

I was almost a little bit nervous, am I going to get it kind of thing, but then I think I was able to apply the knowledge really well, it went together. So it was, and I think it's almost like my confidence, because I have never done this before, but my confidence of my capability to do it went up so I think near the end when I was debugging it I was like oh, I understand what needs to be done for this code to run. Felt a lot better than in the beginning when I was like am I even going to understand this at all?

P5 was concerned that she was going to be unsuccessful, a worrying thought. However, having success and a sense of comprehension allowed her to relax and feel good about her experience.

5.8.4.3 Purpose-first programming was enjoyable, especially when participants were successful

Most participants said that they enjoyed completing the curriculum. *"I think it was really interesting and fun,"* said P1. Interest in coding moderated enjoyment for some participants. For instance, P8 said, *"I enjoyed it, considering I'm not a computer science major or anything and I don't really like this type of stuff."*

Success often influenced enjoyment. *"When I was frustrated I didn't [enjoy it]. But when I was getting things right like in the debugging problem I was like yeah, I really like*

this,” said P5, who struggled with several activities. P2 was the only participant who admitted that she didn’t really enjoy completing the curriculum, but success on problems was a bright spot. She said, *“I don’t really like coding, I don’t really like it but then when I got them right, it was nice.”*

5.8.4.4 Theme summary

Participants typically described feeling successful after completing the curriculum, and even enjoying the experience as a result. Even though the participants were completing highly scaffolded activities that provided a great deal of assistance, participants felt successful because they had a sense of understanding the content and they knew they had solved problems correctly.

5.8.5 Participants felt that purpose-first programming was for beginners and those who need extra help

When asked who would learn best with purpose-first programming, participants described students who were new to programming and students who could not understand code right away. Most participants put themselves in one or both of these categories.

5.8.5.1 Many participants, particularly conversational programmers, had low self-efficacy for programming

Participants often characterized themselves as learners who struggled to understand programming. For instance, P5, a conversational programmer, felt that she was often behind others in class. She said about herself, *“I’m the one who doesn’t get it or I’m in the group of people that’s going slower.”* This feeling of self-efficacy was sometimes related to career choice. P7, a future UX designer who planned to program in her career, said she decided to not become a software engineer because *“I*

don't feel like I have the intellect to become a full-time engineer, and I feel like I don't have enough creative solutions."

Participants were careful to distinguish themselves from the "smart" "hard coders" who would become "backend" engineers. They described certain other computing students as people whose *"mind just sees things like that [plan groupings]"* (P8) or *"who just breeze through stuff"* (P2). By contrast, participants often felt that they needed more opportunities for practice and assistance.

5.8.5.2 Participants felt that plans were best for learners who struggle

Participants characterized purpose-first programming as a fit for people who didn't grasp concepts quickly and needed assistance to succeed. P4 described the curriculum as *"helpful for people who need extra help and practice. Or who may have a harder time understanding concepts."* P2 explained, *"I think it would be useful to everyone but especially for people who maybe are like... I don't want to say slower at learning and need to thoroughly know what each step does in order to understand."* Participants felt that such struggling learners could benefit from the practice opportunities and explanations provided by purpose-first programming.

Many participants felt that learning with plans was a fit for their own needs, since they faced difficulties while learning to code. P5 said, *"I think for someone like me who wants a lot of help, who needs a lot of help to do well, and wants a lot of examples and visuals and documentation, the plans worked really well."* P7 agreed that purpose-first programming provided support she wanted, saying *"I have to fail at the code 10 times before I realize what I'm doing, so I guess I sort of like this."* P1 believed that learning with plans gave a greater opportunity for repetition of concepts than other learning experiences, improving her learning. She said, *"I personally need to practice a lot of time. So I think for someone like me, this would be more helpful."*

5.8.5.3 Participants felt that plans were more appropriate for beginners

Participants often described purpose-first programming as a fit for new programmers and people first learning a topic. P9 said, *"I think plans are pretty useful for beginners like me."* P4 felt that purpose-first programming was more appropriate for novices, saying, *"I think it's better for people who are new to programming."* Describing the process of purpose-first programming, P8 said, *"I think it's just more of a beginner concept."*

P1 described purpose-first programming as the right fit for inexperienced programmers because it was straightforward to understand:

For a non-programmer, I think it's very easy to follow. So someone without any experience or with any intention of doing really complicated stuff in the future, I think it breaks it down really easily. So then it just makes sense logically.

Conversely, participants felt that purpose-first programming may be constraining for people with significant prior knowledge or who were already succeeding at programming. P5 suggested, *"if everybody had to go through the plans it would feel frustrating or maybe even condescending to someone who was an advanced programmer."* P9 felt that these more knowledgeable learners were ready to jump right to applying their knowledge. She said, *"I'm pretty sure people who are more experienced in code who already knew Python or C++ before don't really need plans. They just need practice problems, right?"*

5.8.5.4 Theme summary

Consistent with prior research, participants often expressed low self-efficacy for programming, a characteristic common to conversational programmers [28, 157] and female programming learners [10]. These novices saw a fit for purpose-first programming with their needs, either as learners new to a topic, or learners

who need extra practice and assistance.

5.8.6 Participants believed purpose-first programming gave them conceptual, high-level knowledge

Participants described what they had learned from the curriculum as “basics”, “concepts”, and “familiarity”. For conversational programmers, this knowledge aligned with their goals for learning programming.

5.8.6.1 Participants felt that purpose-first programming helped them learn general, not detailed, knowledge

While participants didn’t believe they could write web scraping code on their own after completing the curriculum, they did feel that they gained a conceptual understanding of web scraping. P9, an aspiring UX designer who plans to program in her career, described her learning outcome by saying, *“I probably won’t remember specifically how to write the exact code line for line....I’ll understand the concepts behind it.”* Similarly, P7, also a future designer, focused on being able to describe content at a high level: *“The next time I talk about web scraping, I’ll definitely be able to define it, know what it is, know... I’m not going to forget the word soup.”*

P8, a future designer who doesn’t plan to program often, felt that the design of purpose-first programming allowed her to focus on key ideas rather than details. She said,

“I liked this because you can focus on it more conceptually without worrying about whether every part of your code is right, which I think helped me understand it more. Seeing which pieces I needed to change and which pieces I could leave alone because I knew it was right.”

P8 described her conceptual understanding as arising from her focus on plan slots and how they are modified.

5.8.6.2 Conversational programmer participants were interested in basic understanding of code.

Conversational programmers felt that learning to code was practical for their career goals, but they desired only knowledge sufficient to understand code, not deep enough to write it. P8 described her learning goals, saying, *"I think just understanding how it works is interesting and just good knowledge for any technology major or occupation. It's just something that makes sense to know. I don't know when exactly I'll use it but I like knowing it."* P2 felt that when talking with software developers, a little bit of coding knowledge would be *"nice to know"*. She also felt that she didn't need to understand code in all its intricacy, saying, *"I don't think I feel like I have to really know it in depth, it's just a good baseline."*

Conversational programmers described the conceptual and general knowledge they gained from purpose-first programming as appropriate for their needs. As someone who wasn't planning on coding in their job, aspiring designer P2 didn't believe that she needed knowledge any deeper than the curriculum provided. Talking about web scraping programs, she said, *"maybe I can't explain it perfectly and explain each part in depth but if I have like a general idea of what it's doing I think that's helpful."* P1, another future designer, described her experience with purpose-first programming as giving her an accessible overview of what code does. She said of her experience with the curriculum, *"it wasn't like I just wrote a random line of code one time, debugged it, and then didn't actually remember how it works....in this curriculum, I actually understand what's going on. And as someone who isn't going to need to program in the future, I just need to understand what's happening, I think that's really helpful."*

5.8.6.3 Theme Summary

Participants felt that purpose-first programming helped them understand code while focusing on concepts. For conversational programmers, this type of curriculum was seen as appropriate for their goals: obtaining a general understanding of coding concepts, not detailed knowledge about how to write code [28, 157].

5.8.7 Participants found curricular content realistic and applicable

Participants found the topic of web scraping "interesting" and "cool". They often described content as "real," because it examples showed "real world" scenarios and utilized "real-life" websites.

5.8.7.1 Participants found web scraping interesting and applicable

The topic of the curriculum appealed to participants, the majority of whom were interested in web technology. This interest motivated them to engage in learning. P6, an aspiring conversational programmers and designer, said, *"It [web scraping] definitely seems like something I'm interested in too, which I feel like just makes me want to learn it."* For P8, the ability to understand why people use web scraping contributed to her interest. She said, *"I thought Web Scraping was interesting, that everything we learned is immediately useful, or we're not able to apply it right but Web Scraping is something that it's really easy to see why it's used, you know?"*

5.8.7.2 Realistic examples reinforced the perception that content was useful and relevant

Participants referenced examples from the curriculum as one reason that the content was realistic. Specifically, the use of familiar websites bolstered their perceptions of authenticity. P9 said, *"I liked using the Cottage Inn and the Rate My Professor websites because they were like real life. Yeah that was cool."* For P5, the examples

illustrated a realistic application of the code. P5 said, *"I also did like the storyline and being able to like oh, there's so many different locations for this pizza place, maybe you want to follow them. Actually making it more of a real world problem I thought was helpful."* This participant appreciated the relatable narrative of the example.

P6 felt that the curriculum showed how to apply code to solve a problem, rather than demonstrating how code worked in a context-free way. She said, *"Using real websites and not only code to show it was also really helpful because just looking at a line of code really means nothing to me if I don't have a visual with it."* The use of real websites also showed participants that the code "really worked." P8 reflected, *"I like that these examples use real websites and SI websites. It was super interesting and you actually go to see that they worked and how they worked."* The content of the curriculum was clearly useful because it worked on familiar websites that participants knew about, not a fake context only for classroom use.

5.8.7.3 Theme summary

Realism and applicability were two reasons that participants valued the curricular content. For participants, a sense of authenticity came from both understanding how code could be applied in the world outside the classroom and confidence that the code they were studying would actually achieve the goals.

5.8.8 Code tracing visualizations are unhelpful if confusing, but useful for deep knowledge if understandable

Participants drew contrasts between purpose-first programming and code tracing activities they had used in prior coursework. These learners had experience tracing code with tracing visualization tools like PythonTutor, Lobster, and debuggers found in IDEs. I identified two perspectives: where program visualization tools are confusing and don't contribute to knowledge, and where program visu-

alization tools are understandable and contribute to deepening knowledge.

5.8.8.1 Some participants found code tracing visualizations difficult to understand and apply, and prefer plans

Some participants had prior experience with PythonTutor [60], a program visualization tool [141] designed to help novice programmers understand the execution flow of programs. When comparing use of PythonTutor to their experience with purpose-first programming, these learners preferred purpose-first programming because they found PythonTutor challenging to understand. *"I mean I guess in theory CodeLens [a PythonTutor variant] is useful but sometimes it gets so confusing,"* said P2. *"Having the plans is helpful because it's like if I know what to do then I can do it, but if I don't know what to do then it's like where do I even start?"*. P2 found purpose-first programming more immediately useful than code tracing visualizations. P5 said, *"CodeLens is a visual thing which I thought would be helpful but honestly I hated it and never used it. And then just from my experience CodeLens was helpful for people who knew programming really well, so it kind of feels like the plans might be for more beginners."* P5 categorized code tracing tools as for non-beginners, while plans were understandable and useful for her current needs.

5.8.8.2 Others found code tracing visualizations to be a helpful method for gaining deeper knowledge, after learning with plans

On the other hand, some participants spoke positively about code tracing visualizations. P9 found the debugger in Visual Studio Code to be a useful tool, and a technique that would help her generalize her knowledge: *"I think plans helped me in the very beginning, but once I understand the concept, debugging is what really helps me further my learning past the plans into more real life scenarios."* During the debugging activity, P1 asked if she could run a debugger on the code, which wasn't possi-

ble. However, reflecting on her problem-problem after the fact, P1 felt that she didn't need to use a code tracing visualization to solve problems in the curriculum. When asked if the curriculum should include a debugger, she said, *"I think the level of detail right now is at a really good spot for someone who is just starting out with web scraping. So I think, yeah, I don't think you need to add anything."*

P4, an aspiring software developer, was in favor of adding a program visualization tool to the curriculum. She described such a tool as *"a little more helpful because you can go step by step through the code and understand what this line does, understand how create this variable, and the information you pull goes into this variable."* P4 was able to understand and make use of code tracing visualization, and felt it would enhance her learning in the curriculum.

5.8.8.3 Theme summary

While all participants felt that plans were useful for immediately applying knowledge and solving problems, participants differed in the role they felt code tracing visualizations should play. For learners who found code tracing visualizations difficult to understand, plans were much preferable. For participants who were comfortable enough to make use of code tracing visualizations for further learning, these technologies could play a role in deepening understanding.

5.8.9 Discussion

I provide preliminary evidence that novice conversational and end-user programmers are motivated to learn with a purpose-first programming approach. Consistent with Eccles' Expectancy-Value theory [41], both a sense of success on purpose-first programming activities and value for purpose-first programming tasks contributed to learners' motivation (see Figure 5.13). Novice conversational and end-user programmers found purpose-first programming activities to be less

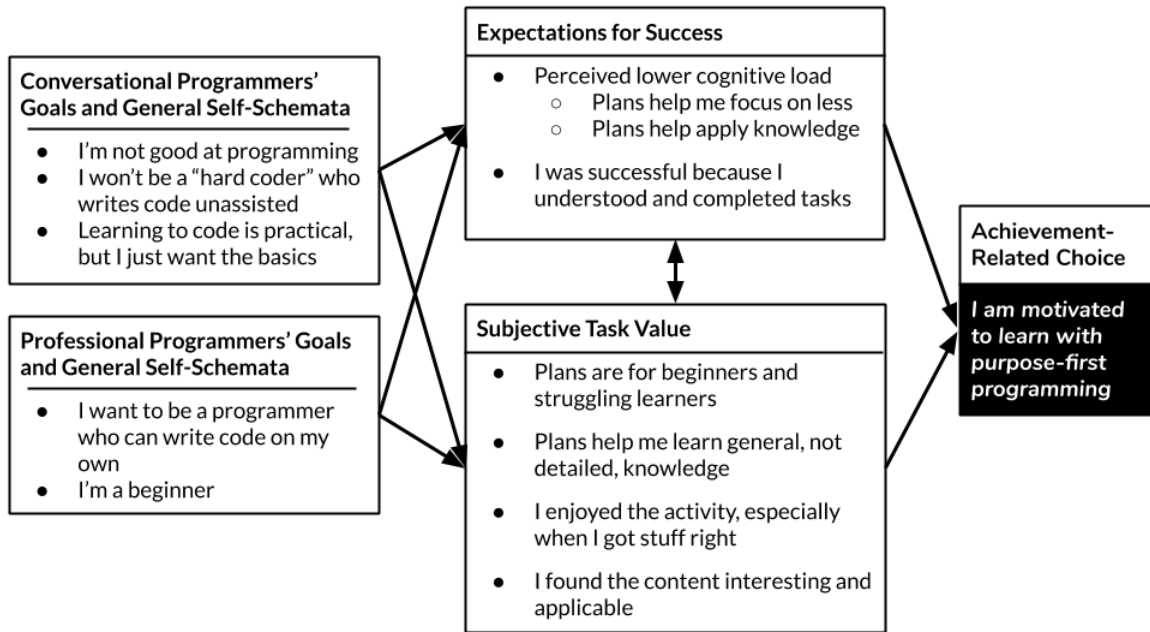


Figure 5.13: The themes mapped onto the Eccles Expectancy-Value Model of Achievement Choice [42].

cognitively difficult than typical programming learning tasks, resulting in a sense of success because they could understand and complete tasks. These learners also found value in purpose-first programming because they enjoyed succeeding on problems and they found the topic interesting and applicable. For some learners, purpose-first programming was valuable because they viewed it as appropriate for their identity as a struggling programming learner. For others, purpose-first programming was valuable because it was a match for their stage as new coders. For conversational programmers in particular, purpose-first programming helped them achieve their goals of a conceptual understanding of code.

Purpose-first programming tasks are much more highly scaffolded than typical programming learning tasks like developing a full program. Despite the high level of support provided by the curriculum, these learners felt successful and accomplished rather than bored or unconfident. This may be explained by the *type* of support that purpose-first programming provides. Gorson and O'Rourke [57] found that novice programmers in introductory courses are particularly likely to

negatively self-assess when they don't know how to start a problem or don't understand a problem statement. Novices are more likely to negatively self-assess in these moments than when they needed to look up syntax or when they get help from others. Participants perceived purpose-first programming as helping them "break down" problem-solving, making the path to a solution easier and clearer. Such support may have helped learners avoid moments when they would be most likely to negatively self-assess.

The identification of plans from "real" programs and experts was a key design element of purpose-first programming intended to bolster authenticity. Participants were informed at the beginning of the study that the plans they would learn in the curriculum were generated from common patterns used by experts. Surprisingly, no participant mentioned this characteristic in their interview. Instead, participants described a sense of authenticity for elements of the curriculum based on what worked in the "real world", such as use of runnable code that "actually worked" to collect data from familiar websites. These novices seemed to value content that achieved a useful task, but they weren't as concerned about using the same coding structures as experts.

Participants related their identities as beginners and people who struggle with programming to their feeling of value for purpose-first programming. This suggests that learners who are confident in their skills and already view themselves as an accomplished programmer may not see value in purpose-first programming. Besides the lack of value, learners who already possess plan schemas would likely find purpose-first scaffolds distracting and cumbersome rather than helpful, according to the expertise reversal effect [77]. Caution should be exercised when using purpose-first programming with populations like computer science majors and advanced students.

5.9 Implications for future curriculum design

The evaluation of this proof-of-concept curriculum provides some insight into how future purpose-first programming learning experiences can be designed to improve learners' experience and motivation.

5.9.1 There are opportunities to streamline activities and reduce user error

New tools designed specifically for purpose-first programming activities could provide more efficient and targeted support than what was possible with the Rune-stone platform. This support may help participants avoid errors they experienced in the proof-of-concept curriculum and achieve a greater sense of success.

The accessibility of plan information is one area where the learner experience could be streamlined. In the proof-of-concept, learners had to visit other ebook pages to view plan information and examples. In a future system, this information could be made available in the editor itself with approaches like pop-up windows or dynamic sidebars. This design may help participants notice more readily when they have formatting errors, which were the most common errors as participants filled the plan slots. This assistance could also make it more evident which plan(s) are the most relevant when editing code, helping learners avoid viewing less relevant plan information.

5.9.2 Clear indication of success can contribute to motivation

Participants were motivated by their sense of success on purpose-first programming activities. Analysis of participant interviews showed that this sense of success came from understanding and completing activities correctly. Frequent interactive feedback in the curriculum made it clear when participants had "gotten it right", likely contributing to their sense of success.

Purpose-first programming activities could be designed without the high level of interactive feedback available in this proof of concept. My findings show that eliminating interactive feedback would risk undermining one of the key contributing elements to learners' motivation. Purpose-first programming provides many opportunities to break down problems into smaller steps. Pairing these small steps with frequent feedback appears to be a recipe for a sense of success, contributing to motivation.

5.9.3 Fading of scaffolding could allow purpose-first programming to be more broadly used

My results suggest that purpose-first programming can be a motivating initial learning experience in a new topic area. The best approach to transition from purpose-first programming to a deeper understanding of code execution remains to be understood, but such a transition is important to meet the needs of non-conversational programmers. While conversational programmers felt that the general, conceptual knowledge they gained from purpose-first programming was sufficient for their needs, learners who planned to program in their career desired the ability to write code on their own and to more deeply understand how code worked.

Incorporating options to fade purpose-first scaffolding may allow a future purpose-first programming system to meet the needs of multiple populations of novice learners simultaneously. In the proof-of-concept, the code writing process was staged across activities, so assembling and tailoring plans took place in separate steps. A dedicated purpose-first programming editor could combine plan assembly and tailoring, offering the ability to drag plan blocks, edit slots, and run code within a single editor. This structured assistance could also be faded as learners progress, or according to their preferences.

5.10 Conclusion

In this chapter, I conceptualized, developed, and evaluated *purpose-first programming*, an approach to programming learning for novice programmers who care more about what code can achieve than how a programming language works. After completing a proof-of-concept purpose-first programming curriculum, novice conversational and end-user programmers were motivated to learn future programming topics with purpose-first methods. This motivation stemmed from a feeling of success, because learners could understand and complete problems, and alignment with goals and self-identity, because learners found the content useful and found the level of support appropriate for their needs. These learners were able to utilize purpose-first supports to complete scaffolded programming activities in a new topic area after only a short period of instructional time.

My findings provide initial support for the effectiveness of purpose-first programming as a motivating starting point for conversational programmers and certain other novice programmers during programming learning. This work connects cognitive theories to theories of motivation in order to present a new approach to programming learning, designed specifically for novices who care more about the opportunities to use code than the operation of programming languages. As aspiring conversational programmers and end-user programmers study programming in greater numbers, the need for a different instructional approach is more apparent. Purpose-first programming can provide a new pathway to programming learning, designed with both the cognition and motivation of these learners in mind. This work opens new avenues to computing, inviting a broader group of learners to engage with programming, particularly those who are less likely to find code semantics motivating.

CHAPTER VI

Summing Up and Looking Forward

6.1 Contributions of this thesis

6.1.1 Novices don't trace code because it's cognitively challenging and has low value

Why would novice programmers choose to not trace code as they problem-solve, a strategy often correlated with success [88, 33]? The first contribution of this thesis is a cognitive and affective explanation for why novices don't trace code. The cognitive complexity of close tracing is a challenge for some learners (Chapter III), who often perform searches for code's goals and patterns instead of close tracing. Some learners also have a low value for the task of code tracing itself (Chapter IV), viewing it as in opposition to their beliefs that they shouldn't have to think like the computer or think like a programmer.

I found that cognitively, carefully tracing through code's execution is time-consuming and potentially overwhelming for some novices. I found that novice learners often search for the *goals* of code rather than trying to trace, and they reason about code's results based on those goals. Some novices value knowledge about *what code can do* more than knowledge about *how code works*. I found that such learners can relate this belief to their choice to not trace code.

A wide variety of programming learners share this preference for a focus on code's applications rather than its syntax or logic, including conversational programmers and end-user-programmers [157, 36]. My findings suggest that this population of programming learners may be unmotivated to complete the code tracing activities common in introductory programming courses. In my analysis of theories of instruction (Chapter II), I found that code tracing is often recommended early in the sequence of instruction, even before activities like understanding or writing full programs [163, 86]. For learners who have a low value or expectancy of success for code tracing, programming learning starts with an activity they are unmotivated to complete.

As the numbers of conversational programmers and end-user programmers grow [21, 127], addressing their particular learning needs will become more important. If learners aren't motivated to trace code, or even reject tracing activities, how can they progress in programming learning that starts from a focus on tracing? Is there another way?

6.1.2 Learning domain-specific programming plans can motivate both conversational and end-user programmers

The second contribution of this thesis is the creation of a curriculum that leads to success for these kinds of programmers. To create a new programming learning pathway for novices who care most about what code achieves, I developed a learning approach called *purpose-first programming* that focuses on domain-specific code plans (Chapter V). To support learners' ability to reason about code without the need for tracing, purpose-first programming scaffolds the ability of learners to apply *plan knowledge*. Rather than using tracing to understand code, novices can use information about the goals common code patterns achieve, and how patterns should be modified to achieve those goals.

The purpose-first programming approach is likely to motivate conversational programmers and end-user programmers because it addresses issues of both expectancy of success and value. Purpose-first programming allows novice learners to quickly create and understand code that achieves goals in a domain of interest, which aligns with the content these learners value [157, 36]. Applying plan knowledge is less complex than performing code tracing, because the learner is working with larger chunks [53] of information. This decrease in cognitive load was expected to increase learners' expectancy of success.

I implemented the purpose-first programming approach in a proof-of-concept curriculum in the domain of web scraping and evaluated it with novice programmers who had lower expectancy of success or value for code tracing, and who were planning to become conversational programmers and end-user programmers. These learners were able to use plan knowledge, such as information about how to tailor plans and information about plan goals and subgoals, to create and understand programs that accomplished tasks. I found that these novices were motivated to continue learning with the purpose-first programming approach. They found purpose-first programming tasks less demanding than other programming learning activities, and felt that the approach was a fit for their learning needs and their identities.

6.1.3 Considering *both* cognition and value can lead to effective approaches

In my investigation of the reasons novices don't trace code, it became evident that cognitive challenges as well as affective challenges contributed to learners' choices. Realizing this, I drew on Expectancy-Value theory [41] to explain learners' motivation for tracing tasks. This theory proposes that motivation is a result of both expectancy of success on a task, and the value for that task. Expectancy and value can influence each other, and both are influenced by goals and self-beliefs.

This thesis is an example of research and design that considers both the cognitive and sociocultural aspects of a learning experience. Purpose-first programming motivates learners by changing the cognition they need to perform to complete programming tasks, and by aligning tasks with the values of learners' communities of practice [85]. Thinking with *plan knowledge* both decreases learners' cognitive load and supports a focus on code's purpose, which aligns with certain learners' values and goals. Thus, purpose-first programming is a "success" (as I described in the second contribution statement) in both cognitive and affective terms.

This approach avoids some pitfalls of work that focuses on either cognition or values exclusively. In Chapter II, I noted that the theories of programming instruction that I analyzed did not consider learners' motivations for tasks. These arguments for ideal instructional orderings are based on inferences about how knowledge of one skill may increase success on another skill. This perspective doesn't address how varied motivation for skills may suggest differentiated learning pathways.

On the other side of the spectrum, instructional designs that center learners' values typically draw on sociocultural approaches that focus on community-level dynamics rather than an individual's cognition [59]. In the constructionist vision, complex tasks, such as creation of e-textiles, provide the opportunity for integration of a wide variety of learner interests and values into programming tasks [112]. In practice, these approaches often require extensive expert support to help learners manage the cognitive challenges of creating a complex artifact with new technology [5]. The third contribution of this thesis is a theoretical lens that considers both cognitive and affective explanations for student challenges in learning programming. This lens has predictive power, since the approach that I designed based on this theory was successful.

6.2 Future work

This thesis found some promising initial results for the ability of a purpose-first programming approach to motivate conversational and end-user programmers. However, there are many research directions yet to explore.

6.2.1 What learner characteristics are correlated with a benefit from purpose-first programming?

In the proof-of-concept, participants were selected based on characteristics that theoretically made them likely to be relatively more motivated by purpose-first programming. However, it's unclear how each of these learner characteristics contribute to motivation. For learners who aren't interested in the particular topic area of a purpose-first programming module, will the activity still be motivating? For learners who are computer science majors, would the highly-scaffolded style of purpose-first programming be demotivating? Exploring the responses to purpose-first programming activities from a wider variety of learners would provide a deeper understanding of motivation.

6.2.2 How can purpose-first programming activities fit into an instructional sequence?

My findings in this thesis suggest that purpose-first programming activities may work best as an initial activity in a new topic area, where it can serve as a motivating introduction that lays a groundwork for later skills. However, purpose-first programming activities must be studied in combination with other learning tasks to make a recommendation about this ordering.

6.2.2.1 Can purpose-first programming activities improve self-efficacy?

In this thesis, I worked with Expectancy-Value theory as one of my primary theoretical frameworks. Expectancy-Value theory was appropriate for my focus on learners' motivation for a single task: code tracing or purpose-first programming. As I move forward to explore how success on purpose-first programming may motivate future activities and self-beliefs, the construct of *self-efficacy* may be more appropriate. Self-efficacy is an individual's judgment of their ability to perform a task [7], and in computing, self-efficacy is a major predictor of future success [118].

Fortunately, self-efficacy is highly manipulable. One method of improving self-efficacy is through "mastery experiences," where a learner understands that they have been successful in a task [8]. I found that purpose-first programming gave novices an opportunity to feel successful while creating and understanding code in an domain of interest. Could purpose-first programming activities provide a "mastery experience" that improves self-efficacy for other programming tasks going forward?

6.2.2.2 How can learners build on a foundation of plan knowledge?

While the learners I studied felt that they would retain general knowledge about web scraping after the activity, I did not measure learning explicitly. What sort of knowledge do learners gain from a purpose-first programming experience, and is this knowledge beneficial for later completion of traditional programming learning activities, like writing code from scratch?

6.2.3 What technology can support development of purpose-first programming curricula at scale?

The creation of the proof-of-concept curriculum required time-consuming identification of plans and development of activities. For the purpose-first program-

ming approach to reach a wide variety of students, technological supports may be needed to lessen the load of curricular development.

6.2.3.1 What platform could support purpose-first programming curricula in a variety of domains?

Purpose-first programming could be applied in any scenario where domain-specific knowledge about programming is valued over knowledge of syntax and semantics. The development of a platform where users could easily build their own purpose-first programming curricula would enable a wide variety of users to create purpose-first programming activities for their particular needs. This platform could support plan-based activity types like those described in this thesis. If provided with sufficient plan information, the purpose-first programming platform could even automatically generate practice problems populated with plan combinations.

6.2.3.2 Can data mining methods identify plans?

The generation of a new set of plans for each domain area is a limiting factor to the adoption of purpose-first programming. If data mining approaches could be developed to support the identification of plans in a domain, and the way those plans can be combined into programs, the effort to build purpose-first programming activities could be reduced. One might imagine a process of data mining a repository of code from a given domain to identify common code patterns, and then generating potential plans and plan combinations for human review and annotation with subgoals and goals.

My findings suggest that drawing plans from a corpus representative of expert practice may not be as important to learners' sense of authenticity as confirmation that plan code works on realistic tasks. This suggests that plan identification algo-

rithms could optimize for the ability of combinations of plans to solve meaningful problems.

Once plans and plan combinations are identified, the skeleton of a purpose-first programming curriculum is in place. This information could be passed to the platform, forming a pipeline from code corpus to curriculum.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] [n.d.]. Beautiful Soup Documentation. <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>. Accessed: 2020-09-15.
- [2] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013*. Technical Report. ACM Press and IEEE Computer Society Press. <https://doi.org/10.1145/2534860>
- [3] Beth Adelson. 1984. When Novices Surpass Experts: The Difficulty of a Task May Increase with Expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 10, 3 (1984), 483.
- [4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2013. *Compilers: Principles, Techniques, and Tools*. Pearson.
- [5] Morgan G Ames. 2018. Hackers, Computers, and Cooperation: A Critical History of Logo and Constructionist Learning. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–19.
- [6] Anthony Anderson, Christina L Knussen, and Michael R Kibby. 1993. Teaching teachers to use HyperCard: a minimal manual approach. *British Journal of Educational Technology* 24, 2 (1993), 92–101.
- [7] Albert Bandura. 1977. Self-Efficacy: Toward a Unifying Theory of Behavioral Change. *Psychological Review* 84, 2 (1977), 191.
- [8] Albert Bandura. 1995. *Self-Efficacy in Changing Societies*. Cambridge University Press.
- [9] Andrew Begel and Beth Simon. 2008. Struggles of New College Graduates in their First Software Development Job. In *Proceedings of the 39th Technical Symposium on Computer Science Education* (proceedings of the 39th technical symposium on computer science education ed.). Association for Computing Machinery, 226–230.
- [10] Sylvia Beyer. 2008. Predictors of Female and Male Computer Science Students' Grades. *Journal of Women and Minorities in Science and Engineering* 14, 4 (2008).

- [11] Sambasiva R Bhatta and Ashok Goel. 1997. Learning Generic Mechanisms for Innovative Strategies in Adaptive Design. *Journal of the Learning Sciences* 6, 4 (Oct. 1997), 367–396. https://doi.org/10.1207/s15327809jls0604_2
- [12] John B Biggs and Kevin F Collis. 2014. *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.
- [13] John B Black, John M Carroll, and Stuart M McGuigan. 1987. What kind of minimal instruction manual is the most effective. In *ACM SIGCHI Bulletin*, Vol. 17. ACM, 159–162.
- [14] Kirsten Boehner, Janet Vertesi, Phoebe Sengers, and Paul Dourish. 2007. How HCI Interprets the Probes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1077–1086.
- [15] Benedict du Boulay, Tim O’Shea, and John Monk. 1981. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International journal of man-machine studies* 14, 3 (April 1981), 237–249. [https://doi.org/10.1016/S0020-7373\(81\)80056-9](https://doi.org/10.1016/S0020-7373(81)80056-9)
- [16] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
- [17] Virginia Braun and Victoria Clarke. 2012. Thematic Analysis. In *APA Handbook of Research Methods in Psychology* (2 ed.). American Psychological Association, 57–71.
- [18] Burning Glass Technologies. 2016. *Beyond Point and Click: The Expanding Demand for Coding Skills*. Technical Report. <https://www.burning-glass.com/research-project/coding-skills/>
- [19] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (Florence, Italy) (ICPC ’15)*. IEEE Press, Piscataway, NJ, USA, 255–265.
- [20] Teresa Busjahn and Carsten Schulte. 2013. The Use of Code Reading in Teaching Programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli, Finland) (Koli Calling ’13)*. Association for Computing Machinery, New York, NY, USA, 3–11. <https://doi.org/10.1145/2526968.2526969>
- [21] Tracy Camp, W Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambruch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Growth of Computer Science. *ACM Inroads* 8, 2 (2017), 44–50.

- [22] Tracy Camp, W Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: The Mixed News on Diversity and the Enrollment Surge. *ACM Inroads* 8, 3 (2017), 36–42.
- [23] S N Cant, David Ross Jeffery, and Brian Henderson-Sellers. 1995. A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology* 37, 7 (Jan. 1995), 351–362. [https://doi.org/10.1016/0950-5849\(95\)91491-H](https://doi.org/10.1016/0950-5849(95)91491-H)
- [24] John M. Carroll, Penny L. Smith-Kerker, James R. Ford, and Sandra A. Mazur-Rimetz. 1987. The Minimal Manual. *Hum.-Comput. Interact.* 3, 2 (June 1987), 123–153. https://doi.org/10.1207/s15327051hci0302_2
- [25] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. SAGE.
- [26] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.
- [27] Michelene TH Chi. 1997. Quantifying Qualitative Analyses of Verbal Data: A Practical Guide. *The journal of the learning sciences* 6, 3 (1997), 271–315.
- [28] P. K. Chilana, C. Alcock, S. Dembla, A. Ho, A. Hurst, B. Armstrong, and P. J. Guo. 2015. Perceptions of Non-CS Majors in Intro Programming: The Rise of the Conversational Programmer. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 251–259.
- [29] Parmit K. Chilana, Rishabh Singh, and Philip J. Guo. 2016. Understanding Conversational Programmers: A Perspective from the Software Industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (San Jose, California, USA) (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1462–1472. <https://doi.org/10.1145/2858036.2858323>
- [30] Tony Clear, J L Whalley, Phil Robbins, Anne Philpott, Anna Eckerdal, and M Laakso. 2011. Report on the Final BRACElet Workshop: Auckland University of Technology, September 2010. (2011). <http://aut.researchgateway.ac.nz/handle/10292/1514>
- [31] Alan Cooper et al. 2004. *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity*. Vol. 2. Sams Indianapolis.
- [32] Kathryn Cunningham, Rahul Agrawal Bejarano, Mark Guzdial, and Barbara Ericson. 2020. “I’m Not a Computer”: How Identity Informs Value and Expectancy During a Programming Activity. (2020).

- [33] 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*. 164–172.
- [34] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice Rationales for Sketching and Tracing, and How They Try to Avoid It, In Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education. In *Innovation and Technology in Computer Science Education (ITiCSE '19)*, 37–43. <https://doi.org/10.1145/3304221.3319788>
- [35] Johan De Kleer. 1984. How Circuits Work. *Artificial intelligence* 24, 1-3 (1984), 205–280.
- [36] Brian Dorn and Mark Guzdial. 2010. Discovering Computing: Perspectives of Web Designers. In *Proceedings of the Sixth International Workshop on Computing Education Research*. 23–30.
- [37] Brian Dorn and Mark Guzdial. 2010. Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 703–712.
- [38] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. <https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9>
- [39] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an Analysis of Program Complexity From a Cognitive Perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (Espoo, Finland) (ICER '18)*. Association for Computing Machinery, New York, NY, USA, 21–30. <https://doi.org/10.1145/3230977.3230986>
- [40] Robert Dyer, Hoan Anh Nguyen, Hriday Rajan, and Tien N Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.
- [41] Jacquelynne Eccles. 1983. Expectancies, Values and Academic Behaviors. *Achievement and achievement motives* (1983).
- [42] Jacqueline S. Eccles. 2009. Who am I and what am I going to do with my life? Personal and collective identities as motivators of action going to do with my life? Personal and collective identities as motivators of action. *Educational Psychologist* 44, 2 (2009), 78.

- [43] Jacquelynne S Eccles et al. 2005. Subjective Task Value and the Eccles et al. Model of Achievement-Related Choices. *Handbook of competence and motivation* (2005), 105–121.
- [44] Barbara J. Ericson and Bradley N. Miller. 2020. Runestone: A Platform for Free, On-Line, and Interactive Ebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) (SIGCSE '20). Association for Computing Machinery, New York, NY, USA, 1012–1018. <https://doi.org/10.1145/3328778.3366950>
- [45] Barbara J. Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying Design Principles for CS Teacher Ebooks Through Design-Based Research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) (ICER '16). ACM, New York, NY, USA, 191–200. <https://doi.org/10.1145/2960310.2960335>
- [46] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) (ICER '14). Association for Computing Machinery, New York, NY, USA, 35–42. <https://doi.org/10.1145/2632320.2632346>
- [47] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) (SIGCSE '16). Association for Computing Machinery, New York, NY, USA, 211–216. <https://doi.org/10.1145/2839509.2844556>
- [48] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) (SIGCSE '17). Association for Computing Machinery, New York, NY, USA, 213–218. <https://doi.org/10.1145/3017680.3017777>
- [49] Sue Fitzgerald, Beth Simon, and Lynda Thomas. 2005. Strategies that Students Use to Trace Code. <https://doi.org/10.1145/1089786.1089793>
- [50] Edward M Gellenbeck and Curtis R Cook. 1991. An Investigation of Procedure and Variable Names as Beacons During Program Comprehension. In *Empirical Studies of Programmers: Fourth Workshop*. Ablex Publishing, Norwood, NJ, 65–81.
- [51] John S Gero. 1990. Design Prototypes: A Knowledge Representation Schema for Design. *AI magazine* 11, 4 (1990), 26–26.

- [52] John S Gero and Udo Kannengiesser. 2014. The Function-Behaviour-Structure Ontology of Design. In *An Anthology of Theories and Models of Design*. Springer, 263–283.
- [53] Fernand Gobet, Peter CR Lane, Steve Croker, Peter CH Cheng, Gary Jones, Iain Oliver, and Julian M Pine. 2001. Chunking Mechanisms in Human Learning. *Trends in cognitive sciences* 5, 6 (2001), 236–243.
- [54] Ashok Goel, Sambasiva Bhatta, and Eleni Stroulia. 1997. Kritik: An Early Case-Based Design System. *Issues and applications of case-based reasoning in design 1997* (1997), 87–132.
- [55] 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. , 493–501 pages.
- [56] Ashok K Goel, Spencer Rugaber, and Swaroop Vattam. 2009. Structure, Behavior, and Function of Complex Systems: The Structure, Behavior, and Function Modeling Language. *Artificial intelligence for engineering design, analysis and manufacturing: AI EDAM* 23, 1 (2009), 23–35.
- [57] Jamie Gorson and Eleanor O’Rourke. 2020. Why do CS1 Students Think They’re Bad at Programming? Investigating Self-Efficacy and Self-Assessments at Three Universities. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 170–181.
- [58] Ira Greenberg. 2007. *Processing: Creative Coding and Computational Art*. Apress.
- [59] James G Greeno, Allan M Collins, Lauren B Resnick, et al. 1996. Cognition and Learning. *Handbook of Educational Psychology* 77 (1996), 15–46.
- [60] Philip J Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 579–584.
- [61] Mark Guzdial. 1995. Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments* 4, 1 (1995), 1–44.
- [62] Mark Guzdial. 2003. A Media Computation Course for Non-majors. *SIGCSE Bull.* 35, 3 (June 2003), 104–108. <https://doi.org/10.1145/961290.961542>
- [63] Mark Guzdial. 2013. Exploring Hypotheses About Media Computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) (ICER ’13). Association for Computing Machinery, New York, NY, USA, 19–26. <https://doi.org/10.1145/2493394.2493397>

- [64] Mark Guzdial, Michael Konneman, Christopher Walton, Luke Hohmann, and Elliot Soloway. 1998. Layering scaffolding and CAD on an integrated workbench: An effective design approach for project-based learning support. *Interactive Learning Environments* 6, 1/2 (1998), 143–179.
- [65] Christine A Halverson. 2002. Activity Theory and Distributed Cognition: Or What Does CSCW Need to DO with Theories? *Computer Supported Cooperative Work: CSCW: an International Journal* 11, 1 (2002), 243–267. <https://doi.org/10.1023/A:1015298005381>
- [66] Brian Harvey and Jens Mönig. 2010. Bringing “No Ceiling” to Scratch: Can One Language Serve Kids and Computer Scientists. *Proc. Constructionism* (2010), 1–10.
- [67] Matthew Hertz and Maria Jump. 2013. Trace-Based Teaching in Early Programming Courses. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (2013), 561–566. <https://doi.org/10.1145/2445196.2445364>
- [68] Cindy E Hmelo and Mark Guzdial. 1996. Of Black and Glass Boxes: Scaffolding for Doing and Learning. (1996).
- [69] Cindy E Hmelo, Douglas L Holton, and Janet L Kolodner. 2000. Designing to Learn about Complex Systems. *The Journal of the Learning Sciences* 9, 3 (2000), 247–298.
- [70] Cindy Hmelo-Silver. 2004. Comparing Expert and Novice Understanding of a Complex System from the Perspective of Structures, Behaviors, and Functions. , 127–138 pages. [https://doi.org/10.1016/s0364-0213\(03\)00065-x](https://doi.org/10.1016/s0364-0213(03)00065-x)
- [71] Cindy Hmelo-Silver, Rebecca Jordan, Catherine Eberbach, and Suparna Sinha. 2016. Systems Learning with a Conceptual Representation: A Quasi-Experimental Study. *Instructional Science* (09 2016). <https://doi.org/10.1007/s11251-016-9392-y>
- [72] Cindy E Hmelo-Silver, Surabhi Marathe, and Lei Liu. 2007. Fish Swim, Rocks Sit, and Lungs Breathe: Expert-Novice Understanding of Complex Systems. *The Journal of the Learning Sciences* 16, 3 (2007), 307–331.
- [73] Luke Hohmann, Mark Guzdial, and Elliot Soloway. 1992. SODA: A computer-aided design environment for the doing and learning of software design. In *International Conference on Computer Assisted Learning*. Springer, 307–319.
- [74] Mark A. Holliday and David Luginbuhl. 2004. CS1 Assessment Using Memory Diagrams. *ACM SIGCSE Bulletin* 36, 1 (2004), 200. <https://doi.org/10.1145/1028174.971373>

- [75] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing* 13, 3 (2002), 259–290. [https://doi.org/10.1006/S1045-926X\(02\)00028-9](https://doi.org/10.1006/S1045-926X(02)00028-9)
- [76] David A Joyner, Ashok K Goel, and Nicolas M Papin. 2014. MILA–S: Generation of Agent-Based Simulations from Conceptual Models of Complex Systems. In *Proceedings of the 19th International Conference on Intelligent User Interfaces*. ACM, 289–298.
- [77] Slava Kalyuga. 2009. The Expertise Reversal Effect. In *Managing Cognitive Load in Adaptive Multimedia Learning*. IGI Global, 58–80.
- [78] David Kirsh. 2010. Thinking with External Representations. *AI & Society* 25, 4 (2010), 441–454.
- [79] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. <https://doi.org/10.1145/1922649.1922658>
- [80] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. 2012. The Knowledge-Learning-Instruction Framework: Bridging the Science-Practice Chasm to Enhance Robust Student Learning. *Cognitive science* 36, 5 (2012), 757–798. <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1551-6709.2012.01245.x>
- [81] Janet Kolodner. 1993. *What Is Case-Based Reasoning?* 3–31 pages. <https://doi.org/10.1016/b978-1-55860-237-3.50005-4>
- [82] Neill Korobov and Avril Thorne. 2006. Intimacy and Distancing: Young Men’s Conversations about Romantic Relationships. *Journal of Adolescent Research* 21, 1 (2006), 27–55.
- [83] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. *SIGCSE Bull.* 37, 3 (June 2005), 14–18. <https://doi.org/10.1145/1151954.1067453>
- [84] David Landy and Robert L Goldstone. 2007. How Abstract Is Symbolic Thought? *Journal of Experimental Psychology: Learning, Memory, and Cognition* 33, 4 (2007), 720.
- [85] Jean Lave and Etienne Wenger. 1991. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press.

- [86] 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* (Perth, Australia) (ACE '11). Australian Computer Society, Inc., AUS, 9–18.
- [87] Raymond Lister. 2016. Toward a Developmental Epistemology of Computer Programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. Association for Computing Machinery, 5–16. <https://doi.org/10.1145/2978249.2978251>
- [88] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) (ITiCSE-WGR '04). Association for Computing Machinery, New York, NY, USA, 119–150. <https://doi.org/10.1145/1044550.1041673>
- [89] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming. , 161 pages. <https://doi.org/10.1145/1595496.1562930>
- [90] Raymond Lister, Otto Seppälä, Beth Simon, Lynda Thomas, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, and Kate Sanders. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *ACM SIGCSE Bulletin*, Vol. 36. 119–150. <https://doi.org/10.1145/1041624.1041673>
- [91] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. *SIGCSE Bull.* 38, 3 (June 2006), 118–122. <https://doi.org/10.1145/1140123.1140157>
- [92] Lei Liu and Cindy E Hmelo-Silver. 2009. Promoting Complex Systems Learning through the Use of Conceptual Representations in Hypermedia. *Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching* 46, 9 (2009), 1023–1040.
- [93] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) (ICER '08). Association for Computing Machinery, New York, NY, USA, 101–112. <https://doi.org/10.1145/1404520.1404531>

- [94] Liz Lyon and Aaron Brenner. 2015. Bridging the Data Talent Gap: Positioning the iSchool as an Agent for Change. *International Journal of Digital Curation* 10, 1 (2015), 111–122.
- [95] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4, Article 16 (2010), 15 pages. <https://doi.org/10.1145/1868358.1868363>
- [96] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. Programming by choice: urban youth learning programming with scratch. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education* (Portland, OR, USA). ACM, New York, NY, USA, 367–371. <https://doi.org/10.1145/1352135.1352260>
- [97] Jane Margolis and Allan Fisher. 2002. *Unlocking the Clubhouse: Women in Computing*. MIT press.
- [98] Lauren Margulieux and Richard Catrambone. 2017. Using Learners' Self-Explanations of Subgoals to Guide Initial Problem Solving in App Inventor. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) (ICER '17). Association for Computing Machinery, New York, NY, USA, 21–29. <https://doi.org/10.1145/3105726.3106168>
- [99] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) (ICER '12). ACM, New York, NY, USA, 71–78. <https://doi.org/10.1145/2361276.2361291>
- [100] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. 2013. Subgoal labeled worked examples improve K-12 teacher performance in computer programming training. In *Proceedings of the 35th Annual Conference of the Cognitive Science Society*, M. Knauff, M. Pauen, N. Sebanz, and I. Wachsmuth (Eds.). Cognitive Science Society, Austin, TX, 978–983.
- [101] Richard E Mayer. 1976. Some Conditions of Meaningful Learning for Computer Programming: Advance Organizers and Subject Control of Frame Order. *Journal of educational psychology* 68, 2 (April 1976), 143–150. <https://doi.org/10.1037/0022-0663.68.2.143>
- [102] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. 2004. Questions, Annotations, and Institutions: Observations from a Study of Novice Programmers. In *the Fourth Finnish/Baltic Sea Conference on Computer Science Education, October 1–3, 2004 in Koli, Finland*. Helsinki University of

Technology, Department of Computer Science and Engineering, Laboratory of Information Processing Science, Finland, 11–19.

- [103] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A Review of the Literature from an Educational Perspective. *Computer Science Education* 18, 2 (2008), 67–92.
- [104] Jan Meyer and Ray Land. 2003. *Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within the Disciplines*. University of Edinburgh.
- [105] Marvin Minsky. 1974. *A Framework for Representing Knowledge*. Technical Report. Massachusetts Institute of Technology-AI Laboratory.
- [106] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals Help Students Solve Parsons Problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (Memphis, Tennessee, USA) (SIGCSE '16)*. ACM, New York, NY, USA, 42–47. <https://doi.org/10.1145/2839509.2844617>
- [107] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (Omaha, Nebraska, USA) (ICER '15)*. ACM, New York, NY, USA, 21–29. <https://doi.org/10.1145/2787622.2787733>
- [108] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. 2012. Ability to 'Explain in Plain English' Linked to Proficiency in Computer-Based Programming. <https://doi.org/10.1145/2361276.2361299>
- [109] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in Plain English' Questions: Implications for Teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (Raleigh, North Carolina, USA) (SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 385–390. <https://doi.org/10.1145/2157136.2157249>
- [110] Greg L Nelson, Benjamin Xie, and Amy J Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (Tacoma, Washington, USA) (ICER '17)*. Association for Computing Machinery, New York, NY, USA, 2–11. <https://doi.org/10.1145/3105726.3106178>
- [111] Fred G Paas. 1992. Training Strategies for Attaining Transfer of Problem-Solving Skill in Statistics: A Cognitive-Load Approach. *Journal of Educational Psychology* 84, 4 (1992), 429.

- [112] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.
- [113] Miranda C Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (Melbourne, VIC, Australia) (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 93–101. <https://doi.org/10.1145/2960310.2960316>
- [114] Dale Parsons and Patricia Haden. 2006. Parsons Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (Hobart, Australia) (ACE '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163. <http://dl.acm.org/citation.cfm?id=1151869.1151890>
- [115] Alan J Perlis. 1982. Special Feature: Epigrams on Programming. *ACM Sigplan Notices* 17, 9 (1982), 7–13.
- [116] Leo Porter, Mark Guzdial, Charlie McDowell, and Beth Simon. 2013. Success in Introductory Programming: What Works? *Commun. ACM* 56, 8 (Aug. 2013), 34–36. <https://doi.org/10.1145/2492007.2492020>
- [117] Judith Preissle and Margaret D Le Compte. 1984. *Ethnography and Qualitative Design in Educational Research*. Academic Press.
- [118] Vennila Ramalingam and Susan Wiedenbeck. 1998. Development and Validation of Scores on a Computer Programming Self-Efficacy Scale and Group Analyses of Novice Programmer Self-Efficacy. *Journal of Educational Computing Research* 19, 4 (1998), 367–381.
- [119] R Douglas Riecken, Jurgen Koenemann-Belliveau, and Scott P Robertson. 1991. What Do Expert Programmers Communicate by Means of Descriptive Commenting. In *Empirical Studies of Programmers: Fourth Workshop*. Ablex Publishing Corporation, Norwood, New Jersey, 177–195.
- [120] Robert S Rist. 1989. Schema Creation in Programming. *Cognitive Science* 13, 3 (1989), 389–414.
- [121] Robert S Rist. 1995. Program Structure and Design. *Cognitive Science* 19, 4 (1995), 507–562.
- [122] Robert S Rist et al. 1986. Plans in Programming: Definition, Demonstration, and Development. In *Empirical Studies of Programmers*. 28–47.
- [123] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming: Which Concepts Do Students Struggle With?. In

Proceedings of the 2016 ACM Conference on International Computing Education Research (Melbourne, VIC, Australia) (ICER '16). Association for Computing Machinery, New York, NY, USA, 143–151. <https://doi.org/10.1145/2960310.2960333>

- [124] Jorma Sajaniemi and Marja Kuittinen. 2003. Program Animation Based on the Roles of Variables. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (San Diego, California) (*SoftVis '03*). Association for Computing Machinery, New York, NY, USA, 7–ff. <https://doi.org/10.1145/774833.774835>
- [125] Jorma Sajaniemi and Marja Kuittinen. 2005. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education* 15, 1 (2005), 59–82.
- [126] Johnny Saldaña. 2016. *The Coding Manual for Qualitative Researchers* (3 ed.). SAGE.
- [127] Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the Numbers of End Users and End User Programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 207–214.
- [128] Roger C Schank and Robert P Abelson. 1977. *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures*. Lawrence Erlbaum.
- [129] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) (ICER '08). Association for Computing Machinery, New York, NY, USA, 149–160. <https://doi.org/10.1145/1404520.1404535>
- [130] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem Is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) (*Koli Calling '15*). Association for Computing Machinery, New York, NY, USA, 87–96. <https://doi.org/10.1145/2828959.2828963>
- [131] David Williamson Shaffer and Mitchel Resnick. 1999. "Thick" Authenticity: New Media and Authentic Learning. *Journal of interactive learning research* 10, 2 (1999), 195–216.
- [132] Herbert A Simon. 1996. *The Sciences of the Artificial*. MIT Press.
- [133] Elliot Soloway. 1985. From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research* 1, 2 (1985), 157–172.

- [134] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (1986), 850–858.
- [135] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860.
- [136] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (Sept. 1984), 595–609.
- [137] Elliot Soloway, Kate Ehrlich, Jeffrey Bonar, and J. Greenspan. 1982. What do novices know about programming? In *Directions in Human-Computer Interaction*, Andre Badre and Ben Schneiderman (Eds.). Ablex Publishing, 87–122.
- [138] Elliot M Soloway and Beverly Woolf. 1980. Problems, Plans, and Programs. <https://doi.org/10.1145/800140.804605>
- [139] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Aalto University. <https://aaltodoc.aalto.fi/handle/123456789/3534>
- [140] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Transactions on Computing Education* 13, 2, Article 8 (July 2013), 31 pages. <https://doi.org/10.1145/2483710.2483713>
- [141] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education* 13, 4 (2013), 15.1– 15.64. <https://doi.org/10.1145/2490822>
- [142] Juha Sorva and Teemu Sirkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli, Finland) (Koli Calling '10)*. Association for Computing Machinery, New York, NY, USA, 49–54. <https://doi.org/10.1145/1930464.1930471>
- [143] James Clinton Spohrer. 1989. *MARCEL: A Generate-Test-and-Debug (GTD) Impasse/Repair Model of Student Programmers*. Ablex Publishing.
- [144] James C. Spohrer and Elliot Soloway. 1985. Putting it All Together is Hard for Novice Programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Vol. March. IEEE.
- [145] James C Spohrer, Elliot Soloway, and Edgar Pope. 1985. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction* 1, 2 (June 1985), 163–207. https://doi.org/10.1207/s15327051hci0102_4

- [146] James G. Spohrer and Elliot Soloway. 1986. Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers Workshop*, Elliot Soloway and S. Iyengar (Eds.). Ablex, 230–251.
- [147] Leigh Ann Sudol-DeLyser. 2015. Expression of Abstraction: Self Explanation in Code Production. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (Kansas City, Missouri, USA) (SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 272–277. <https://doi.org/10.1145/2676723.2677222>
- [148] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive Science* 12 (1988), 257–285. Theory behind worked examples.
- [149] Donna Teague. 2015. *Neo-Piagetian Theory and the Novice Programmer*. Ph.D. Dissertation. Queensland University of Technology. <http://eprints.qut.edu.au/86690/>
- [150] Donna Teague and Raymond Lister. 2014. Longitudinal Think Aloud Study of a Novice Programmer. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148 (Auckland, New Zealand) (ACE '14)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 41–50. <http://dl.acm.org/citation.cfm?id=2667490.2667495>
- [151] Allison Elliott Tew and Mark Guzdial. 2010. Developing a Validated Assessment of Fundamental CS1 Concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (Milwaukee, Wisconsin, USA) (SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 97–101. <https://doi.org/10.1145/1734263.1734297>
- [152] Lynda Thomas, Mark Ratcliffe, and Benjy Thomasson. 2004. Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (Norfolk, Virginia, USA) (SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 250–254. <https://doi.org/10.1145/971300.971390>
- [153] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in Novice Programmers' Poor Tracing Skills. *SIGCSE Bull.* 39, 3 (June 2007), 236–240. <https://doi.org/10.1145/1269900.1268853>
- [154] Swaroop S Vattam, Ashok K Goel, Spencer Rugaber, Cindy E Hmelo-Silver, Rebecca Jordan, Steven Gray, and Suparna Sinha. 2011. Understanding Complex Natural Systems by Articulating Structure-Behavior-Function Models. *Journal of Educational Technology & Society* 14, 1 (2011), 66–81.
- [155] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research*

Workshop (Berkeley, CA, USA) (*ICER '09*). Association for Computing Machinery, New York, NY, USA, 117–128. <https://doi.org/10.1145/1584322.1584336>

- [156] Lev S. Vygotsky. 1978. Socio-cultural theory. *Mind in society* (1978).
- [157] April Y. Wang, Ryan Mitts, Philip J. Guo, and Parmit K. Chilana. 2018. Mismatch of Expectations: How Modern Learning Resources Fail Conversational Programmers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) (*CHI '18*). Association for Computing Machinery, New York, NY, USA, 1–13. <https://doi.org/10.1145/3173574.3174085>
- [158] David Weintrop and Uri Wilensky. 2015. To Block or not to Block, that Is the Question: Students' Perceptions of Blocks-based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*. 199–208.
- [159] Daniel S Weld. 1983. *Explaining Complex Engineered Devices*. Technical Report. Bolt Beranek and Newman Inc Cambridge MA.
- [160] Jacqueline Whalley, Christine Prasad, and P. K. Ajith Kumar. 2007. Decoding Doodles: Novice Programmers and Their Annotations. In *Proceedings of the Ninth Australasian Conference on Computing Education - Volume 66* (Balarat, Victoria, Australia) (*ACE '07*). Australian Computer Society, Inc., AUS, 171–178.
- [161] Roger D Wimmer and Joseph R Dominick. 2013. *Mass Media Research*. Cengage learning.
- [162] David Wood, Jerome S Bruner, and Gail Ross. 1976. The Role of Tutoring in Problem Solving. *Journal of child psychology and psychiatry* 17, 2 (1976), 89–100.
- [163] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. 2019. A Theory of Instruction for Introductory Programming Skills. *Computer Science Education* 29, 2-3 (2019), 205–253.
- [164] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) (*SIGCSE '18*). Association for Computing Machinery, New York, NY, USA, 344–349. <https://doi.org/10.1145/3159450.3159527>