Qualifying Novices' Conceptual Resources in Computer Science I

by

Carlos Aldana Lira

A thesis presented to the Honors College of Middle Tennessee State University in partial fulfillment of the requirements for graduation from the University Honors College.

Fall 2024

Thesis Committee:

Dr. Tasha Frick, Thesis Director

Dr. Grant Gardner, Second Reader

Dr. Richard Nagorski, Thesis Committee Chair

Qualifying Novices' Conceptual Resources in Computer Science I

by Carlos Aldana Lira

APPROVED:

_____

Dr. Tasha Frick, Thesis Director

Professor, Chemistry

_____

Dr. Grant Gardner, Second Reader

Professor, Biology

_____

Dr. Richard Nagorski, Thesis Committee Chair

Professor, Chair, Chemistry

A mi madre y mi padre

Acknowledgements

I thank my advisor, Tasha Frick, for her support and guidance throughout this past year of research. I also thank Grant Gardner for allowing me to pester him with questions about learning and educational video games and for his continued support since then. I additionally thank Amy J. Ko for risking a summer's worth of time and energy and allowing me to explore qualitative, educational research at the University of Washington and for her continued support in my work. I met each unsure of myself and my ability to conduct educational research, but their unwavering support has led me to where I am now. My work has only been better for their mentorship.

I am also indebted to the many professors and mentors I have had throughout my undergraduate career. My journey towards this thesis and my research interests have been long and winding, and many have influenced that journey's trajectory. In this vein, I thank Andrew Fialka, Alfred Lutz, April Weissmiller, Rimal Ramchandra, Katy Hosbein, Todd O'Neil, and Carla Fox. Each have sparked or supported my research interests in their own way, and the absence of any one would make my work worse.

I thank the computer science professors and students without whom this work would have been impossible. I also thank the Interactive Media department for allowing me to borrow the audio/video equipment necessary to conduct the study.

Finally, this project was supported by an Undergraduate Research Experience and Creative Activity (URECA) grant through the Office of Research and Sponsored Programs at Middle Tennessee State University.

Abstract

Learning can be conceptualized as change in long-term memory, but this change is mediated by the concepts, strategies, and values learners bring into the classroom. This thesis describes an ontology of the prior knowledge novice programmers enter introductory computing courses with and how such knowledge mediates their programming practice. To generate this ontology, I conducted 1:1 clinical interviews with 6 novice programmers recruited from introductory programming courses at Middle Tennessee State University. During the interviews, I presented novices with a series of programming problems and asked them to think aloud as they reasoned about them. I found that novice programmers entered their programming courses with manifold conceptions of the assignment operator and conceptual resources about knowledge of programs. This thesis clarifies the difficulties novices have when learning to program, contributes a computing-specific description of novices' knowledge, and provides groundwork on which future design-based research may be conducted.

Contents

List of Tables

List of Code Listings

List of Terms

Computer Physical hardware receiving, storing, processing, and outputting data. Though consumer devices like desktops, laptops, or phones are certainly computers, examples also include devices *embedded* in other hardware — like cars, smart home devices, fridges, videogame consoles, microphones, or full-body scanners.

Computing The discipline and community studying the technical, human, and social aspects of computation, computers, programs, software systems, and software engineering. Also known as *computer science*.

Programming The act and broad practice of designing, implementing, and evaluating computer programs, often by designing, writing, and reading program text in specialized programming languages. Also known as *coding*.

Code The sequence of written statements instructing a computer how to read, write, process, or transform data, often written in terms of a programming language with well-specified syntax and semantics like C++ or Python.

CHAPTER I: Introduction

Our society exists in computing. Our work, learning, communication, and play is mediated by computation, and the rate at which computing is reshaping these activities is increasing. Further, computing lies at the center of immediate and future controversies, such as applied artificial intelligence, automation of work, social and filter bubbles through social media algorithms, and rampant data collection (Ko et al., 2020). Respectively, these trends have led to renewed interest to teach computing to cultivate a technologically literate workforce (Noonan, 2017; U.S. Department of Labor, 2007) and to a citizenry capable of reasoning and making decisions about computational phenomena (Blikstein & Moghadam, 2019). These populations need, at least, conversational fluency in how computational technologies are designed, implemented, and evaluated. That is, everyone should, at least, know how to program.

Although everyone should be fluent in programming, learning to program appears difficult. Computing educators have repeatedly observed students rarely leave Computer Science 1 (CS1) with the design and problem-solving skills necessary for programming (Robins et al., 2003). Similarly, across a multinational, multi-institutional assessment, McCracken et al. (2001) found most post-CS1 students were unable to solve ostensibly skill-appropriate programming problems, and many did not know how to begin formulating a solution for programming problems. So great is the difficulty some instructors have sought a "programming gene" or assessments determining whether prospective programmers are "worth" teaching (Dehnadi & Bornat, 2006; McCartney et al., 2017; Vivian et al., 2014). Further, though pass-fail rates are not alarmingly low (Bennedsen & Caspersen, 2007; Watson & Li, 2014), these observations suggest a need for significant improvement if CS1 is to leave students able to apply the knowledge we hope to impart to them.

Of course, learning to program *is* complex. The concepts required to "know" a particular programming language are precisely defined, demanding learners attain a dense, self-referential set of concepts to understand even the simplest programs (Robins, 2010). Similarly, the task of programming requires learning the orthogonal skills of reading and writing individual program statements and generic, self-contained program plans or schemata (Xie et al., 2019). Fluidity in *problem-solving* with programs not only depends on learners' development of such generic programming schemata (Sweller, 1988), but additionally requires declarative knowledge, strategies, and models for *designing* programs, additionally demanding fluency in program syntax, program semantics, and the problem domain (Robins et al., 2003). *Debugging* programs also requires troubleshooting strategies, summative code comprehension, and hypothesis generation (McCauley et al., 2008). Finally, the task of learning these components is mediated by barriers potentially imposed by the computing environment (Ko et al., 2004), and achieving general fluency in programming requires integrative skills for orchestrating and composing such components into a coherent practice (Huang et al., 2024).

However, complexity does not mean learning to program has to be difficult. Luxton-Reilly (2016) has suggested that instructors simply expect students to learn more than they reasonably can in an introductory course, as most students reach CS1 learning goals by the end of the subsequent course in the learning sequence, CS2. Rather, if instructors' learning goals are to be feasible for novices, instructors should meet novices where they are. At scale, aligning curricula with novices' extant knowledge will require definitional work: Pears et al. (2007) observed no single consensus for "learning to program" in the CE literature, instead seeing broad goals of learning a programming language, attaining problem-solving skills, or knowing how to design software systems emerging from it. In parallel, aligning curricula demands *describing* what novices know and how they *apply* their knowledge to programming, as such a description would ground assumptions for what can and cannot be taught within a course.

Part of knowing where novices are is examining how are their knowledge differs from experts'. Computing education researchers have described a litany of *misconceptions* novices appear to develop or bring into CS (Qian & Lehman, 2017). For example, Pea (1986) observed novices believe the computer continually evaluates 2 or more lines of code in parallel when it really evaluates code sequentially, 1 line at a time. However, while indexing such misconceptions enables CS instructors to identify, prevent, and correct them (Lewis et al., 2019), focusing on only novices' misconceptions neglects ways in which their prior knowledge may be productive for programming or constructive for expertise (Danielak, 2019). In fact, continuity in form and function between novice and expert knowledge is a central tenant of constructivist theories of learning, requiring analysis of novices' knowledge and development (Smith III et al., 1994).

Further, most applications of learning theory in computing education research are shallow (Szabo & Sheard, 2022). Not only does this suggest opportunity for future work, it is reflected in the community's haphazard use of the term "misconception", such as in reference to errors in reasoning, episodic mistakes, syntax or semantic errors, or, as properly defined, deeply rooted conceptions misaligned with computing canon (e.g., du Boulay, 1986; Oliveira et al., 2023; Pea, 1986; Qian & Lehman, 2017). Moreover, in merely indexing such mistakes without significant connection to learning theory, misconceptions-focused work deprives the computing education research community of the model-based theories required of the field as an engineering, technological, and social science (Tedre & Pajunen, 2022). Similarly, though indices of novices' mistakes suggests some topics are more difficult than others, disconnect from theory prevents such work from being actionable for the design and analysis of novel learning environments (Nelson & Ko, 2018).

To better understand student conceptions, computing educators need an account of novices' prior knowledge — from within or without the discipline of computing — as they enter the classroom and how they apply such knowledge in their programming practice. Here, *knowledge* includes knowledge of what *is* and of *how* to do something, explanations

for how phenomena occur, and conceptions of a discipline. *Practice* includes programmers' actual behavior when engaged in the discipline of programming, such as how they design, implement, and verify programs or software systems. Armed with accounts of novices' prior knowledge and practice, the computing education community may begin to investigate how novices' resources may support expert-level programming practice.

1.1 Research Questions

A theory of novice programmers' conceptual resources stands to benefit from prior work. Drawing from contemporary literature, Robins et al. (2003) summarized the relationships between novices' knowledge and the constituent tasks of programming in **Table 1.1**. The framework suggests programming is not a unitary activity but instead composed of three distinct tasks: designing an algorithm to solve a problem, generating syntax-complaint code to solve the problem, and evaluating the program to determine its correctness. Novices may bring different knowledge, strategies, and models to bear at each phase. Under this framework, novices' conceptual resources may be initially understood as the knowledge, strategies, and models they bring into classrooms and their practice as program design, generation, and evaluation.

Building off of the proposed framework, this thesis describes a novel ontology of novice programmers' knowledge relevant to program design, generation, and evaluation and how novices use this knowledge in similar problem contexts. In particular, I answer the following research questions:

RQ1 At Middle Tennessee State University, what knowledge, strategies, and models do novice programmers possess at the start of CS1?

RQ2 Of the identified knowledge, strategies, and models, how do novice programmers use them when designing, generating, and evaluating Python or C++ programs?

	Knowledge	Strategies	Models
Design	of planning methods, algorithm design, formal methods	for planning, problem solving, designing algorithms	of problem domain, notional machine
Generation	of language, libraries, environment / tools	for implementing algorithms, coding, accessing knowledge	of desired program
Evaluation	of debugging tools and methods	for testing, debugging, tracking / tracing, repair	of actual program

Table 1.1: A programming framework summarizing the relationships between the knowledge, strategies, and models demanded in the programming phases of design, generation, and evaluation. In general, distinctions between knowledge kinds and programming activities are fuzzy (Robins et al., 2003).

Resultantly, this thesis contributes a knowledge analytic framework through which to understand novices' programming behavior and suggestions for how novices' knowledge supports expert-level programming practice. Through this framework, computing education instructors may better locate novices in their learning trajectories and identify proficiencies to capitalize on and effect positive learning outcomes.

CHAPTER II: Literature Review

This chapter outlines the background of this thesis and related work. Summarily, I describe computing education researchers' description of novice programmers' misconceptions, ways in which this work has and does not serve the computing education community, and its relation to science and mathematics education.

2.1 Novice Conceptions

Mature disciplines have a *canon* of knowledge. Biologists, chemists, and mathematicians see, think, and talk about phenomena in ways accepted by their respective communities. Educational institutions transmit disciplines' canonical knowledge through instruction to nurture new disciplinary experts, but learners carry or develop alternative conceptions through their life experiences. For example, when asked what forces act on a ball at the apex of a toss, a physics student may answer that two forces act: the impetus delivered by the tosser's hand and gravity. In contrast, the physicist answers that only one force acts: gravity (diSessa, 2006). With respect to physics canon, the student is *wrong*. In particular, the student has a *misconception* of force, one liable to produce incorrect explanations or support additional misconceptions. For learning, misconceptions can represent barriers to learning new content, such as when the misconception contradicts new information, or barriers to effective practice. As barriers to practice, misconceptions inhibit learning new strategies for engaging in a discipline and afford opportunities for failure — potentially raising additional, affective barriers to learning.

Computing education researchers have described a litany of misconceptions that novice programmers develop while learning to program (Qian & Lehman, 2017). Much of this work attributes misconceptions to novices' natural language knowledge and expectations. L. A. Miller (1974), for example, reported novices confused the Boolean operator `and` for

the operator `or`, speculating that novices mistakenly projected their semantics from everyday procedure specification. Similarly, du Boulay (1986) reported novices readily attribute real-world semantics to English keywords like `then`, `and`, or `repeat`, leading to misinterpretations of program semantics. Bonar and Soloway (1985) have claimed such natural language knowledge to be a significant source of misconceptions, arguing that such misconceptions emerge as novices "patch" their programming knowledge with everyday knowledge of procedure specification.

Other work has speculated natural language knowledge interferes with learning to program in subtler ways. Pea (1986) argued that novices infer a set of programming language-independent misconceptions by programming *as if* the computer were an intelligent conversational partner with interpretative powers. Similarly, C. S. Miller and Settle (2019) observed novice programmers frequently referred to *properties* of an object in place of the object itself (e.g., referring to `name` for `object.name` in place of `object`) and contended this resulted from novices slipping into the natural language habit of referring to an object closely related to a referent instead of the referent itself. However, Taylor (1990) has argued that such application of natural language discourse can variably result in "superstrategies" or "superbugs" when programming, such that natural language knowledge is not bad wholesale. For example, use of natural language may represent a superstrategy when students successfully reason about program inputs and behavior in natural language, enabling translation of their understanding into formal program syntax. However, use of natural language may represent a superbug when it tacitly introduces assumptions about program input, behavior, or output, such as the belief that a computer can execute lines of code out-of-sequence and in parallel.

More recent work has documented novices' misconceptions beyond the context of natural language. For example, Swidan et al. (2018) found that novice Scratch programmers believed that variables can hold more than one value, that assignments could store unresolved equations, and that a false, single-branch condition would terminate the program,

among others. For database programming, Miedema et al. (2021) found novice SQL programmers exhibited misconceptions thought to emerge from four knowledge sources:

- **Prior courses**: For example, novices believed the mathematical symbol $\neq$ was a valid operator in SQL, and others believed the syntax == formed a comparison operator in SQL, as it does in other programming languages like Python or C++.
- **Generalizations**: For example, novices incorrectly reused code or mistakenly wrote syntax that worked in some contexts but not others.
- **Natural Language**: For example, novices expected `IS NOT` to be a valid comparison operator in SQL or used synonyms of SQL keywords.
- **Deficient mental models**: For example, novices simply did not know how certain features of the language behaved, leading to incorrect conceptions of the language or program.

However, identified misconceptions ranged from conceptual misunderstandings, such as believing a statement's scope to be larger than it is, to mistakes attributable to forgetfulness, issues related to English-language learning, or deviant problem-solving approaches. Similarly, other research continues to use *misconception* to refer to mistakes or slips. Oliveira et al. (2023), for example, analyzed snapshots of students' Java code before and after refactoring and identified 25 "refactoring misconceptions," such as failing to simplify `if` statements containing more than one condition, not updating a variable reference when converting a `for` loop to a `for-each` loop, or erroneously replacing a `for` loop with a `for-each` loop. Broadening the term "misconception" out from conceptions contrary to a discipline's canon, as the term was defined above, inhibits our ability to rigorously study and design for students' understandings, as well as for the cognitive processes distinct from misconceptions, such as apparent forgetfulness.

Other work has examined student conceptions as they are or examined how task features contribute to the emergence of misconceptions. Xinogalos (2015), for example, ex-

amined novice Java programmers' conceptions of *classes* — written specifications for collections of data and procedures — and *objects* — instantiations of classes — finding that most understood classes as descriptions of objects and objects as models of real-world phenomena. In contrast, Herman et al. (2012) identified many misconceptions students exhibit while translating English problem specifications to Boolean statements but found that such misconceptions emerged from variations in task features rather than misunderstandings. Similarly, Chao et al. (2018) found that while students exhibited incorrect mental models of recursion — schemata for how programs can "call" themselves, creating a new, running instance of the program — students presented with tasks providing greater contextual support more often constructed normative mental models of recursion, suggesting students can recognize and use task features to construct conceptions on-the-fly.

Computing education researchers have called for additional research into student misconceptions. Clancy (2004), for example, has called for further research into *why* novices develop misconceptions of language syntax and advanced language constructs, such as iterators, generic functions, or templates. Similarly, Lewis et al. (2019) called for further research to better arm instructors with expectations of what students might miss, accounting for instructors' expert blind-spots. However, this research is conducted at the risk of merely pointing out the differences between novices and experts and neglecting the potentially productive knowledge novices enter the classroom with (Danielak, 2019). Equally, its general methodology lends itself to clouding the difficult cognitive and epistemological work novices engage in (Danielak, 2022). We additionally risk mistakenly separating computing and everyday ways of thinking, potentially neglecting goals of computational thinking and development of the whole learner in favor of the narrow slice presented through their work. This is in spite of prior research recommending researchers examine how novices' conceptions might be leveraged or how they develop into expert-level conceptions (Ben-Ari, 2001; Qian & Lehman, 2017).

Finally, merely misconceptions-focused research does not advance our interests as designers and educators. This is because misconceptions alone describe *what* is wrong with students' ideas but give little insight into *how* they have come to be wrong and how they might *serve* designs of learning environments — beyond their being rooted out, at least. Summarily, misconceptions do not provide a robust *model* of student thinking that computing education researchers can use as scientists and designers. That is, they do not provide abstract accounts of how students think about programming that allow researchers or practitioners to explain, predict, or understand that thinking. As an amalgam of engineering, technology, and social science research, computing education theory has drawn from various research paradigms, but Tedre and Pajunen (2022) have found it more fruitful to draw from developing *models* of phenomena rather than grand theories. Further, for the purposes of design, models outline design spaces within which novel learning environments and instructional strategies might be devised (Nelson & Ko, 2018). Bonar and Soloway (1985) have shed insight into how misconceptions are generated, but gaps remain, such as how prior knowledge interacts with program design and ways in which such knowledge might be productive for program generation.

2.2 Novices' Conceptions are Transitory and Context-Sensitive

Catalogs of novices' misconceptions enables instructors to understand, prevent, and correct them, but they serve designers of learning environments poorly by providing an incomplete picture of programmer cognition and tacitly summarizing learners' conceptual resources as unproductive, incorrect, or harmful to learning. To account for this, this thesis draws from learning research in science and mathematics education.

2.2.1 Novices' Conceptions are in Transition

Novices *become* experts; they transition. Theories of learning attempt to describe how this transition occurs at varying levels of abstraction. For example, *constructivism* primarily

contends learners construct new knowledge in terms of prior knowledge by either assimilating new information into their body of knowledge or accommodating new information by reorganizing their body of knowledge. *Cognitivism* contends learning is a change in long-term memory and mediated by a series of relatively independent information processors. Constructivist theories of learning foreground how old pieces of information interact with new ones; cognitivist theories foreground how information processors mediate perception, retrieval, and action (Doroudi, 2021)

Conceptual change is a theory of learning foregrounding how *concepts* develop. This is done to distinguish between surface-level kinds of learning, such as memorization or mere assimilation of information, and the kind of restructuring necessary to think about or perceive situations in new ways (diSessa, 2006). Several studies in science and mathematics education have modeled how conceptual change occurs. Building from Piaget's theory of assimilation and accommodation, Posner et al. (1982) contended conceptual change occurred as learners became dissatisfied with known concepts and perceived new concepts as intelligible, plausible, and fruitful. Additionally, Posner et al. suggested students' prior knowledge, metaphysical beliefs, and epistemological commitments influenced whether conceptual change occurred as well. Pintrich et al. (1993) began to extend Posner et al.'s earlier argument, contending that models of conceptual change should account for learners' goals, interests, and motivations and that students tend to not actively seek coherency as suggested.

While this work charted out the litany of factors that influence deep forms of learning, it neglected descriptions of precisely *what* was changing at levels of granularity greater than "assimilation" or "accommodation." diSessa and Sherin (1998) examined contemporary conceptual change literature and concluded much of it had an unclear ontology of conceptual change, saying little about *how* concepts change. In response, the authors proposed that a theory of concepts should describe what they are, what it means for a student to "know" them, the qualitative distinctions that exist between concepts, the processes through which

they change, and how long they take to change. Further, Smith III et al. (1994) contended that much work examining differences between novices and experts mistakenly prescribed that novices' knowledge should be rooted out, neglecting that novices' ways of knowing and intuitions are continuous in either content or form with experts' conceptions. Subsequent conceptual change work has analyzed students' conceptions and change through various lenses, such as through (Duit & Treagust, 2003):

- **Epistemological status**: How students' concepts change in response to a concepts' intelligibility, plausibility, and utility.
- **Ontological status**: How the *what* of students' concepts change, such as how their understanding of heat may change from *fluid-like substance* to *kinetic energy*.
- **Perception of models**: How students understand models in relation to reality, such as whether they are merely representations or 1:1 relations.
- **Contextual use**: How students' conceptions of phenomena change between contexts.
- **Affect**: How conceptual change is influenced by affective and metacognitive factors.

Conceptual change research, then, seeks to describe *what* students know and proposes accounts of the *processes* by which that knowledge changes. Within this tradition, a significant division has emerged between those who believe that students' concepts form coherent theories of knowledge and those who believe that students' concepts form networks of fragmentary knowledge (diSessa, 2006). For example, diSessa (1993) proposed novice physics students' knowledge partly consists in disconnected intuitions for what kind of physical phenomena can occur and how they might be explained. Similarly, Hammer and Elby (2003) proposed students' beliefs and processes about knowledge and knowing consist in similarly small, context-sensitive resources. However, recent work has prescribed abandoning the coherence question in favor of investigating the circumstances in which novices draw upon

their conceptual resources to *construct* coherent frames of reference and *process* information, broadly proposing that situations bear *conceptual dynamics* influencing how frames of reference are constructed (Sherin et al., 2012). That is, it may be more productive to ask *when* and *why* novices construct coherent ways of thinking about phenomena than to determine if the underlying conceptions are, in general, coherent or fragmented.

2.2.2 Novices' Cognition is Context-Sensitive

The preceding section has explained that knowledge claims of students' conceptions must have a particular ontology and at least entertain how such concepts might change. Subsequent work has further problematized discussion of novices' misconceptions by contending that the deployment of conceptions is *contextual*. In particular, *competence* is enabled by the cognitive context in which a learner acts and is *defined* by the social context in which they reside (Brown et al., 2015). Resultantly, this work contends claims about novices' knowledge must account for context, as whether novices' knowledge is productive — constructive of competence — varies with context. This work resides within the tradition of knowledge analysis, which generally contends that knowledge is made up of discrete, interacting units in a conceptual ecology and that the deploying of those units is contextual (diSessa et al., 2015). The discrete units are *constructed* from prior knowledge, but their use and application are mediated by *cognitive* structures processing and storing information.

Context-sensitivity is important for misconceptions research in two ways. First, it suggests that novices may possess canonical knowledge but simply do not perceive the situation as one demanding that knowledge, resulting in failure to transfer or apply knowledge across contexts. As a result, an explanation for a misconception cannot merely be that the learner is confused; the explanation must account for how the learner perceives the situation and the factors influencing that perception (Levin et al., 2013). Second, it suggests that novices may possess appropriate knowledge but variably deploy it. As a result, an explanation for a misconception cannot merely be that the student is deficient; it must attend to the possibility

that a feature of the task situation did not elicit that knowledge, similar to the previous point. In turn, context-sensitivity suggests that learning is, in part, regulation of information recall and impressing upon the learner the importance and relevance of normative concepts.

Some work in this line of conceptual change research and cognition exists in computing education. Chao (2012), for example, identified that novices intuit that computers are actors producing results, potentially generating flawed models of program recursion. Similarly, Chao et al. (2018) found that features of a task influence the mental model of recursion that students construct. In concert, these findings describe how novices' conceptual resources influence their understanding of how programs behave when executed, particularly programs with recursion. Additionally, Lewis (2012) contended how students' construct interpretations of *program state* may be influenced by conceptions of what features of a program context are *relevant* to program state. These conceptions were thought to be influenced by prior knowledge and tractable to change. Finally, Danielak (2019) suggested that context-sensitivity implies that some misconceptions result from the misapplication of ordinarily useful knowledge, requiring that computing education researchers take greater care in describing students' knowledge and begin from the assumption that the knowledge deployed has a use in students' contextual understanding.

2.3 Summary

In this chapter, I surveyed the misconceptions novice programmers encounter while learning to program. I discussed how documenting these misconceptions is useful but neglects the complete picture of programmers' cognition and the conceptual resources novices bring to bear on programming. I summarized bodies of work in science, mathematics, and computing education that model novices' conceptual resources and how they mediate novices' transition into experts.

This review's intent is not to deprecate misconceptions as an instructional or design tool. As stated above, indexing the missteps novice programmers might make when learning

to program enables the understanding, prevention, and correction of such missteps (Lewis et al., 2019). However, I hope it is clear that too great a focus on novice programmers' mistakes neglects the rich conceptual resources they enter classrooms with and the arduous work they engage in while programming. Further, while it would be correct to observe that a programming language's keywords relation to English words or task feature's relation to students' in-the-moment mental models are small details, it is exactly these small details that learners are most sensitive to. As designers and educators, our duty is to remain sensitive to these details despite their size.

CHAPTER III: Methodology

In this chapter, I describe the conceptual framework, context, and methods of the study described in this thesis. Summarily, I recruited and screened participants from three sections of an introductory computer science course at Middle Tennessee State University. I then interviewed six participants and qualitatively analyzed the resultant transcripts and video to identify conceptual resources at work in their programming knowledge and practice. I begin with my conceptual framework because it is inextricably linked to my methods.

3.1 Conceptual Framework

As suggested in **Section 2.2**, my conceptual framework begins from the premise that *learning* is *change* in discrete mental objects. For example, if we compare a student's conception of *plant* before and after an introductory course in biology, we might say they *learned* if their concept appears to have changed to accommodate ideas about structures — such as cells, tissues, or organs — and processes — such as cellular respiration, photosynthesis, or growth. We might say a chemistry student has *learned* if their conception of *solution* appears to have changed to accommodate procedural ideas, such as dimensional analysis and dilution. In general, if we let a student's initial conception of a phenomenon be A and their later, canonical conception of something be B, then productive learning is change of concept A to concept B.

However, this formulation of learning, alone, says little about how novices appear competent in some problem contexts and incompetent in others. Competence suggests they possess canonical concept B, but incompetence suggests that continue to possess misconception A. Why do novices use B in some contexts but A in others? Further, this formulation fails to specify *what* a concept is. Is a concept a category of things, belief about the world, coherent theory of phenomena, or a model? Leaving these questions unanswered does not

mean we cannot observe, measure, facilitate, or even design for learning, but it certainly limits the *granularity* at which we can do so.

As suggested in **Section 2.2**, a line of research in the learning sciences has argued that novices' conceptions are transitory, and their application is context-sensitive. Here, I adopt a theory of learning and cognition extending from that body of literature, *Knowledge in Pieces*, that claims *knowledge* is made up of many dynamic concepts and *competence* emerges from the coordinated activation of those entities. The ontology of the concepts are diverse, and their activation is contingent on what the learner perceives or judges salient in their context. Learning remains, in part, change in concepts. However, it is additionally change in the *processes* of activating and applying networks of concepts to competently problem-solve or make sense of the world. In summary, learning is change in *networks of concepts* at work in learners' minds, and cognition is viewed as the result of those networks' dynamic activation and change.

I adopt this theory of learning partly for the theoretical considerations above but also for pragmatic purposes. KiP is *cognitive* insofar as it supposes learners' attention to features of the problem environment activate conceptual resources existing in either memory or at an intuitive level. Activated resources frame learners' interpretations of the problem context and ultimately structure their sensemaking strategies. KiP is also *constructive* insofar as it supposes learners' prior knowledge has a *genesis* in which it was first conceived, likely in a context in which it was a useful, and *changes* at either a resource or system level to accommodate new experience. Analyses of novice programming stand to benefit from KiP's cognitivist tendencies, as programming is an explicit mode of problem-solving, but also from its constructivist tendencies, as learning to programming implicates prior knowledge.

3.2 Classroom Context

I conducted this study in Fall 2024, drawing participants from three sections of Computer Science I (CSCI 1170) in the College of Basic and Applied Sciences at Middle Tennessee

State University (MTSU). I selected CSCI 1170 because it is MTSU's designated introductory programming course for Computer Science majors and non-majors; it appeared likely participants would have little experience with programming, providing a suitably novice population for this study.

The class content of the three sections of CSCI 1170 — sections A, B, and C — are as follows. Section A delivered instruction and assignments in the C++ programming language, and sections C and B in the Python programming language. Section A's content followed Gaddis (2018); sections B and C's content followed Gaddis (2020). All students attended 3 hours of lecture and 1 hour of laboratory sessions every week for 15 weeks. In general, instruction was lecture-based and supplemented with program demonstrations or live programming. **Table 3.1** summarizes the course content for the first 5 weeks of the semester, indicating novice participants knew only the C++ or Python syntax and semantics necessary for simple programs with input and output processing and control structures.

Week No.	**Course Content**
1	Syllabus Review, Introduction to Computers, Programming, and UNIX
2	Input and Output Processing and Language Basics
3	Decision Structures and Boolean Logic
4	Logical Operators, Boolean Variables, and Nested Decision Structures
5	Repetition Structures

Table 3.1: The content covered in the first five weeks of CSCI 1170, by week.

3.3 Participant Recruitment and Screening

3.3.1 Recruitment Methods

For each section, I recruited participants during a lecture session in the third week of the semester. I briefly explained the purpose, methods, risks, and benefits of this study to the students as a class, and I then distributed informed consent forms and surveys to interested students. Participants were offered a $20 Amazon gift card for their participation in an interview. After allowing students up to 10 minutes to read and complete the informed consent form and survey, I collected the materials and left the classroom.

The survey collected students' name, age, major, minor, and contact information and assessed students' prior programming experience by five self-report questions. Question 1 asked students to select one of five options describing how they first learned how to program; the last option permitted a free response if the rest did not capture how they learned how to program. Question 2 asked students to describe any programming projects they may have written outside of their computer science courses. Questions 3 and 4 asked students to describe their proficiency in the Python and C++ programming languages by selecting one of four options: "Not at all", "Somewhat", "Slightly", and "Very". Question 5 asked students to report any other languages they are proficient in and describe their proficiency as either "Not at all", "Somewhat", "Slightly", or "Very". The full survey is listed in **Appendix 1.1**.

3.3.2 Selection Methods

Following collection of students' written surveys, my advisor and I applied the screening rubric listed in **Appendix 1.2**. The rubric excluded participants who appeared to be intermediate or expert programmers, leaving only apparent novices in the interview pool. This ensured interviewed participants were obliged to draw from either their new or out-of-domain knowledge when performing interview tasks, better positioning me to induct such knowledge resources in their talk.

Excluded participants were characterized by having significant programming experience, completed programming projects requiring concepts taught after CS1, or having appreciable proficiency in Python, C++, or any other programming language. After applying the exclusion rubric, we ordered the remaining participants in the order in which they were most "novice-like" using the rubric listed in **Appendix 1.3**. Novices were characterized by having no prior programming experience, not having completed any programming projects, and indicated no proficiency in Python, C++, or any other programming language.

After sorting participants from most novice-like to least novice-like, I contacted the first 6 novice-like participants by email. If a student replied within 3 days of receiving the email, I scheduled them for an interview on a first-come, first-serve basis. If a student failed to reply within 3 days, I notified them they would be not be selected for an interview and contacted the next most novice-like student in the list. This continued until I successfully scheduled 6 participants for an interview. **Table 3.2** lists interviewed participants by pseudonym, major, programming language, and week interviewed. Because all participants were interviewed in at least the fourth week of the semester, I assumed they were familiar with basic syntax, input and output processing, and conditional and looping control structures (see **Table 3.1**).

Participant	Major	Language	Week Interviewed
S0	Computer Science	C++	4
S1	Actuarial Science	Python	4
S2	Computer Engineering	C++	4
S3	Computer Science	C++	4
S4	Environmental Science	C++	4
S5	Environmental Science	C++	5

Table 3.2: Information for interviewed participants.

3.4 Data Collection

My objective was to elicit participants' talk in programming contexts to infer knowledge, strategies, and models productive to program design, generation, and evaluation (see **Table 1.1**). To accomplish this, I conducted task-based, clinical interviews with 6 programming novices from 3 sections of CSCI 1170 at MTSU. The interviews were approximately 1 hour in length and audio- and video-recorded. I collected participants' scratch-work or notes as data as well.

3.4.1 Interview Protocol

I conducted 6 task-based, clinical interviews. *Task-based* denotes the interviews revolved around participants' solving and thinking about a set of programming problems, described below. *Clinical* denotes that my conduct attended to each participants' unique mode of thinking. This meant that I continually attempted to reframe problems, introduce new opportunities for sensemaking, or ask clarifying questions to make participants' thinking most visible (diSessa, 2007). In this way, clinical interviews encourage analysts to generate novel characterizations of participants' thinking rather than evaluating their thinking or verifying existing models of it (Clement, 2000).

During the interviews, I restricted myself to presenting the programming problems and asking probing questions to participants. For example, in response to participants' reasoning, I asked "Could you talk more about that?" or "What do you mean by that?" If participants asked me questions to verify their answers, I responded with questions like "Why do you think that?", "How did you come to think that?", or "What do you think?" If a participant was silent for an appreciable amount of time, I prompted them to continue thinking aloud by asking, "What are you thinking about?" or "What are you considering?" Similarly, if participants finished a problem without elaborating on their thought process, I asked questions like "How did you think through that?" or asked them to explain their process as they might to a classmate.

If participants were making visible errors while solving the problem, I did not attempt to correct them. For example, while most participants demonstrated misconceptions around variable assignment, they are expected to know variable assignment after Week 2 or 3 of the semester, so I did not correct them. However, as the interviews were conducted in Weeks 4 and 5 of the semester, some students were not familiar with `while` loops, which are necessary to complete the problems (see **Table 3.1**). In these cases, I stopped the participant and briefly explained `while` loops in terms of preceding programming problems or the immediate problem. If participants continued to misinterpret `while` loops after this explanation, I did not attempt to correct them.

3.4.2 Interview Tasks

During the interviews, I administered up to 5 programming tasks and asked participants to think aloud as they solved them. Each task was designed to elicit the kind of knowledge characterized in one or more of the programming phases shown in **Table 1.1**, as summarized in **Table 3.3**. To ensure these tasks were appropriate for participants' knowledge, I presented the tasks to the CS course instructors from which participants were to be recruited. Based on their feedback, I revised the tasks.

Problem	Design	Generation	Evaluation
1			✓
2			✓
3		✓	
4	✓	✓	
5	✓		✓

Table 3.3: Alignment of interview tasks with phases of programming described in **Table 1.1**. A checkmark ("✓") indicates the task is intended to elicit the knowledge, strategies, and models deployed in either programming design, generation, or evaluation.

The remainder of this section describes each task in detail for the C++ programming language. I describe the C++ variants because 4 of the 5 interviewed participants learned CSCI 1170 content in C++. Readers familiar with programming may prefer to read the tasks in **Appendix B**, which lists the programming tasks as they were presented to students during the interviews.

Problem 1: Swapping Two Variables

Problem 1 asks participants to describe what the program in **Listing 3.1** does. The program swaps the values of two variables. In line 1, the program first makes commands defined in a file named `iostream` available to the programmer. These commands include `std::cin` and `std::cout`, which allow the programmer to receive input from the user and output text to the user, respectively. In lines 4 and 5, the program initializes two variables of integer type named `a` and `b`. Lines 7 and 8 set the values of variables `a` and `b` to the first two numbers the user submits when running the program.

```
1  #include <iostream>
2
3  int main() {
4      int a;
5      int b;
6
7      std::cin >> a;
8      std::cin >> b;
9
10     int c = a;
11     a = b;
12     b = c;
13
14     return 0;
15 }
```

Listing 3.1: The C++ variant of Problem 1. This code is accompanied with the prompt "Consider the below program. What does the program do?" and is designed to elicit the kind of knowledge required for program evaluation (see **Table 1.1**). The program receives two numbers as input, stores them in variables `a` and `b`, and swaps the values of the variables.

The significant content of Problem 1 is contained in lines 10, 11, and 12. In C++ and Python, the assignment operator (=) sets the *value* of the variable on the left-hand side to the *value* of the variable on the right-hand side at the time of assignment. For example, let the value of `a` be 5 and the value of `b` be 7. Then, line 10 sets the value of the newly declared integer variable `c` to 5, and line 11 sets the value of `a` to 7. However, the value of `c` remains 5, even *after* line 11 executes. Line 12 sets the value of `b` to 5. When the program ends at line 14, the values of `a`, `b`, and `c` are 7, 5, and 5, respectively. The values of `a` and `b` have *swapped*.

Problem 2: Finding the Maximum Value

Problem 2 asks participants to describe what the program in **Listing 3.2** does. The program finds the largest integer in a stream of integers submitted by the user. Like **Listing 3.1**, the program begins by including commands needed for program input and output and by declaring integer variables, namely `x` and `number`. The role of the variable `x` is to record the value of the largest integer encountered in the input stream. The role of the variable `number` is to record the integer most recently submitted by the user. Line 7 sets the value of `number` to the first integer submitted by the user.

The significant content of Problem 2 is contained in lines 8–14. Line 8 begins a `while` loop that will terminate if and only if the value of `number` is set to -1. Rather, the commands in lines 9–13 will repeat in sequence until the user submits a value of -1 in response to the prompt emitted on line 13. Line 9 tests if the value of `x` is less than `number`. If so, line 10 will execute, setting the value of `x` to the current value of `number`. In effect, lines 8–14 update the value of `x` only if the user submits a value larger than all previously submitted values. Thus, line 16 prints the *maximum value* encountered in the stream.

Problem 3: Translating Pseudocode to Program Syntax

Problem 3 asks participants to write C++ or Python code that is semantically equivalent to the pseudocode listed in **Listing 3.3**. Pseudocode is a description of an algorithm in approx-

```
1  #include <iostream>
2
3  int main() {
4      int x = 0;
5
6      int number = 0;
7      std::cin >> number;
8      while (number != -1) {
9          if (x < number) {
10             x = number;
11         }
12
13         std::cin >> number;
14     }
15
16     std::cout << x;
17
18     return 0;
19 }
```

Listing 3.2: The C++ variant of Problem 2. This code is accompanied with the prompt "Consider the below program. What does the program do?" and is designed to elicit the kind of knowledge required for program evaluation (see **Table 1.1**). The program receives a stream of integers until the user inputs -1 and records the largest number encountered along the way.

imately natural language. The pseudocode describes an algorithm to compute the factorial of a user-submitted value `N`. **Listing 3.4** lists a C++ implementation of the pseudocode.

The significant content of Problem 3 is contained in lines 9–11 of **Listing 3.3**, a loop. Line 9 specifies the condition for the loop, stipulating that lines 10 and 11 are to repeat until the value of `J` is greater than the value of `N`. Line 10 assigns the variable `P` a new value: the previous value of `P` multiplied by `J`. In effect, if the value of `P` and `J` in the kth iteration of the loop is p_k and j respectively, then the value of `P` is given by $p_k = p_{k-1} \times j$. Line 11 simply adds 1 to the value of `J`.

To implement the pseudocode in lines 9–11 of **Listing 3.3**, the programmer must map the condition specified in line 9 — that the loop terminates when the value of `J` is greater than `N` — to an equivalent condition suitable for a control structure in either C++ or Python. For example, in **Listing 3.4**, the condition is mapped to the equivalent condition that the loop executes while `J` is less than or equal to `N` (i.e., `J <= N`). Possible errors include lin-

```
1  read in a number N
2
3  initialize number P
4  set P equal to 1
5
6  initialize number J
7  set J equal to 1
8
9  repeat the following until J is greater than N:
10     set P equal to P multiplied by J
11     increment J by 1
12
13 print P
```

Listing (3.3) The pseudocode given for Problem 3. This code is accompanied with the prompt "Implement the below algorithm as a program" and is designed to elicit the kind of knowledge required for program generation (see **Table 1.1**). The program computes and prints the factorial of a user-submitted value `N`.

```
1  #include <iostream>
2
3  int main() {
4      int N;
5      std::cin >> N;
6
7      int P = 1;
8      int J = 1;
9
10     while (J <= N) {
11         P = P * J;
12         J++;
13     }
14
15     std::cout << N;
16
17     return 0;
18 }
```

Listing (3.4) The C++ implementation of Problem 3. The program computes and prints the factorial of a user-submitted value `N`. `J++` is equivalent to `J = J + 1`.

early mapping the condition, by writing `J > N`, or specifying only a less-than condition, by writing `J < N`; a program containing either of these errors will not produce the factorial of `N`. The remaining lines, lines 10 and 11, were expected to be trivial mappings.

Problem 4: Soloway's Rainfall Problem

Problem 3 asks participants to write C++ code satisfying the below prompt.

> Design and implement a program reading integers from standard input until the value 99,999 is encountered. Print the average of the integers encountered before 99,999 to standard output. Ensure your program works for all inputs.

In effect, the prompt asks participants to implement a C++ variant of Elliot Soloway's Rainfall Problem, first formulated in Soloway (1986). **Listing 3.5** lists a C++ solution. Implementing the program requires recalling and composing many templates for accomplishing common programming goals, such as running a sentinel-controlled loop (line 11), keep-

ing a running sum and count (lines 12–3), and guarding against division by zero (line 19). Failure in formulating these goals and composing requisite templates will result in an erroneous program, such as one that does not terminate when 99,999 is encountered, includes 99,999 in the average, or divides by zero. As such, the prompt is thought to elicit the kind of knowledge required for program design, generation, and — if the programmer is astute — evaluation (see **Table 1.1**).

```
1  #include <iostream>
2
3  const int SENTINEL = 99999;
4
5  int main() {
6      int sum = 0;
7      int count = 0;
8      int number = 0;
9
10     std::cin >> number;
11     while (number != SENTINEL) {
12         total += number;
13         count++;
14
15         std::cin >> number;
16     }
17
18     int average = 0;
19     if (count > 0)
20         average = total / count;
21
22     std::cout << "Average:␣" << average << std::endl;
23
24     return 0;
25 }
```

Listing 3.5: The C++ solution to Problem 4. The program repeatedly asks a user to input a number until the integer 99,999 is entered. The program then checks if the user entered at least one number. If so, the program prints the integer average; otherwise, it prints zero.

Problem 5: Evaluating a Solution to Soloway's Rainfall Problem

Problem 5 asks participants to evaluate the correctness of **Listing 3.6**, which is framed as a response to the prompt in Problem 4. The code successfully keeps a running sum and count of the integers submitted by the user and avoids dividing the sum by zero, but the program also includes 99,999 in its average, failing to meet the prompt. Most of its commands are similar to those in previous problems, but I briefly describe the two newcomers: line 3 declares the *constant* integer value 99,999 under the identifier `SENTINEL`; line 10 initializes a *Boolean* variable `no_sentinel` that can be either `true` or `false`. In effect, the `while` loop repeats the commands in lines 13–22 in sequence until the user submits the value 99,999.

The program's error is primarily contained in lines 17–21. The program checks the value of `number` *after* adding it to the running sum `total` and incrementing the value of `count`. So, the average will always include the value of 99,999. This can be fixed by moving lines 17–21 between lines 13 and 14. However, this must be accompanied by an `if` statement checking if the value of `count` is greater than 0 to guard against division by zero in line 24.

3.5 Data Analysis

I began data analysis by generating and cleaning transcripts of the interviews. I generated transcripts using Zoom's transcription feature and cleaned them by listening to the recordings, attributing speakers to statements in the process. Participants' pseudonyms were used in place of their names in the transcripts.

After transcription, my thesis advisor and I analyzed the data by inducting and deducting patterns to infer productive knowledge, strategies, and models for programming. We first independently read through the data and assigned preliminary labels to participants' statements or utterances. These labels are called *codes* and summarize salient properties of the data, such as participants' conception of a programming construct. We then met to discuss our readings and analyses of the data, reconciling our codes and any disagreements

```cpp
#include <iostream>

const int SENTINEL = 99999;

int main() {
    int total = 0;
    int count = 0;

    int number = 0;
    bool no_sentinel = true;

    std::cin >> number;
    while (no_sentinel) {
        total += number;
        count += 1;

        if (number != SENTINEL) {
            std::cin >> number;
        } else {
            no_sentinel = false;
        }
    }

    int average = total / count;
    std::cout << "Average:␣" << average << std::endl;

    return 0;
}
```

Listing 3.6: The C++ variant of the code for Problem 5. This code is accompanied with a prompt explaining that a student, Aubrey, wrote the program in response to the prompt in Problem 4 and concluding with "Why is Aubrey's program correct/incorrect?" The problem is designed to elicit the kind of knowledge required for program evaluation (see **Table 1.1**). The primary error of the code is including 99,999 in the average.

encountered. We then repeated this process over many iterations to converge on a common set of *themes*, which often subsume and summarize sets of codes identified in previous coding runs (Cohen et al., 2017; Fereday & Muir-Cochrane, 2006). This method summarily organizes and reduces the wealth of information in qualitative data, enabling reliable, clear interpretations and characterizations of the phenomena described (Tenenberg, 2019).

We did not calculate statistical agreement metrics but instead recorded and resolved disagreements, following Hammer and Berland (2014)'s description of qualitative coding as tabulations of claims about the data and outcomes of analytic processes. Accordingly,

we opted for thick description of our results rather than statistical metrics. If disagreements are pertinent to my presentation of the data, I describe them with the data below.

3.6 Positionality

I disclose my identities and experiences here to afford the reader insight into how these aspects of myself may have shaped the design and results of this thesis. I am a male Guatemalan-American born in the United States to two immigrant parents. I grew up in a bilingual household and in the same geographical region as Middle Tennessee State University (MTSU). Unlike the participants in this study, I spent most of my youth programming videogames in software systems like GameMaker: Studio, LÖVE2D, and Godot. Thus, I entered CSCI 1170 already knowing the content to be taught, so my position relating to experiences in CSCI 1170 and learning to program differed substantially from participants'.

Additionally, I am a Computer Science major at MTSU. I have completed CSCI 1170 with one of the professors whose class I have sampled from, and I am therefore familiar with the content, lecture, and homework of CSCI 1170 and subsequent coursework. I often disclosed this fact to participants to assure them I was familiar with their coursework and experiences. I additionally have experience tutoring students in undergraduate Chemistry and Biology courses, and I often worked with students in these courses to identify and extend their prior knowledge.

CHAPTER IV: Results

The purpose of this research was two-fold. First, I sought to identify the conceptual resources computer science novices enter CS1 classrooms with. Second, I aimed to understand how novices applied these resources when programming. This chapter begins by describing the conceptual resources participants appeared to use when reasoning about program design, generation, and evaluation. I describe each resource and hypothesize contexts in which they appeared useful to participants in their learning. The chapter then describes how participants used these resources in their programming practice. There, I present relevant fragments of the transcripts and describe my analysis.

4.1 RQ1: What Knowledge, Strategies, and Models do CS1 Novices Possess?

Participants appeared to possess many conceptual resources for designing, generating, and evaluating programs. The prevalence and uses of these resources waxed and waned across problem contexts, suggesting problem features differentially elicited participants' resources. Here, I describe the general content and properties of the identified resources one-by-one, but I describe their use in greater depth in the following section. To help denote what is and is not a resource, I italicize all resources as I refer to them. **Table 4.1** summarizes the identified conceptual resources.

4.1.1 *Assignment as Equality*

This resource is active when learners interpret the assignment operator as an *assertion* of mathematical equality rather than an assignment of a *value* to a *variable* or *identifier*. Resultantly, learners appear to believe assignment operands must taken on a shared value; it is suggested that if the operands do not, the computer will throw a run-time or compile-

Resource	Description	Example
Assignment as equality	Assignment asserts an equality between the values of its two operands.	" . . . Whatever I put in for `a` is going to be equal to `b` [. . .] If `c` is equal to `a`, then these two are gonna be the same. So, I'm assuming this is just saying that all of these are going to be the same number?"
Assignment as change	Assignment changes the value of an operand to the value of the other operand.	"We started the program with the variables' value being 0, but that does not mean it's permanently changed for the entire code. We can change that along the way."
Assignment as transitive	Assignment links the value of an operand to the value of the other operand.	"And it will show the variable `x`, and `x` will equal to the number which was entered for the variable `number` on line 13."
Identifier as value	An identifier directly represents a value.	"And that would make 10 equalling 12 . . . "
Knowing is incremental	Knowledge about a program is built token-by-token or line-by-line.	"I don't see any syntax errors. I mean, if the plus equals (+=) means what I think it means, then it looks like every variable is defined, the condition is defined, doesn't look like there's any syntax errors."
Knowledge is authoritative	Knowledge comes from either the professor or the computer.	"Well, if it was somebody in my class, I would just be like, '[Professor A] said during a spe- cific lecture, specifically during our math expressions lecture, [they] explained what each of these operators means, so like the equal sign, the double equal sign, and then [they] went through and explained that the exclamation point is a not operator, so it's saying it's not equal to in this specific instance."'

Table 4.1: The knowledge resources identified during participants' design, generation, and evaluation of programs. A resource is not necessarily a proposition but instead an intuition

time error. Importantly, participants applied and abandoned this resource without explicitly referring to it. They often simply adopted an alternative conception of the assignment statement as the interview progressed, suggesting this resource works at the level of intuition rather than explicit application.

I hypothesize *assignment as equality* is transposed from learners' prior knowledge in mathematics and reinforced by successful applications in introductory programming. The assignment operator in the C++ and Python programming languages have the same form as statements of equality in mathematics. For example, the C++, Python, and mathematical statements `int a = 12;`, `b = "foo"`, and $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$ have the same form $\square = \square$. Thus, the three statements can be interpreted as having the same *symbolic form* or semantics (Sherin, 2001). In particular, all three statements may be interpreted as an assertion that the left-hand side of the operator is equal to the right-hand side.

4.1.2 *Assignment as Change*

This resource is active when learners interpret the assignment operator as *changing* the value of its left-hand operand to that of the right-hand operator. For example, if the variables `a` and `b` are set to `7` and `12` respectively, learners whose *assignment as change* resource is active will believe `a` takes on the value of `12`. Implicitly, the value of `a` is asserted to change across time. I hypothesize this resource is learned as students become familiar with their programming language's sequential model of execution.

Two details of this resource are significant. First, like *assignment as equality*, the resource is applied and abandoned without any verbal indication, suggesting it lies at the level of an intuition rather than a recognized rule about the assignment operator. Second, activation of this resource does not necessarily lead to a canonical model of the assignment operator. Activation of other knowledge resources may augment the expected behavior of the assignment operator, such as expecting it to link variables across time. Further, absence

of knowledge, such as about type inference and compatibility, may lead learners to coerce variables' types inappropriately.

4.1.3 *Assignment as Transitive*

This resource is active when learners interpret the assignment operator as *linking* the values of two variables across time. Resultantly, learners appear to believe the statement `a = b` links `a` to `b` such that any change in the value of `b` is reflected in the value of `a`. This resource requires that learners first interpret the assignment operator as changing the value of its left-hand operand. Thus, I hypothesize *assignment as transitive* requires activation of *assignment as change*. Hence, activation of *assignment as change* does not guarantee expert-level performance.

4.1.4 *Identifier as Value*

This resource is active when learners *replace* variables' identifiers with their values. For example, if the variables `a` and `b` have values `13` and `37` respectively, learners whose *identifier as value* resource is active will believe the statement `a = b` is equal to `13 = 37`. I hypothesize this resource is often activated in concert with *assignment as equality*, as its activation supports interpreting the statement `a = b` as an erroneous statement of equality, but it is distinguished here to accommodate instances in which learners simply confuse the assignment operator (`=`) with the Boolean equality operator (`==`). In such cases, it is possible for a learner to correctly interpret `a` and `b` as identifiers of regions in computer memory whose values are `13` and `37` but misinterpret the assignment operator as comparing those values.

Like *assignment as equality*, I hypothesize *identifier as value* has its genesis in learners' prior mathematical knowledge. In particular, the mathematical form $x = \square$ suggests that x can be replaced with whatever occupies the space $\square$ in any equation x is in. As a result, the C++ or Python forms `a = 10;` and `b = "foo"` readily lend themselves to the belief that `a` and `b` can simply be replaced with `10` and `"foo"`. Further, as this conception

is *productive* in C++ forms like `cout << "Average " + average;`, instructional content may inadvertently reinforce this resource's use.

4.1.5 *Knowing is Incremental*

This resource is active when learners construct a line or program's meaning token-by-token or line-by-line respectively. This resource is partly an operationalization of the code comprehension behavior expected among novices when compared to experts, but I intend for this to also account for the *criteria* learners use to design, generate, and evaluate programs. In particular, the fitness of a program to a specified design or the behavior of a program is known by linearly aggregating the semantics of individual tokens or lines rather than considering the program as a whole. Successfully attributing semantics, independent of their correctness, to each token or line is sufficient to "know" a program.

I hypothesize this resource is learned as students become familiar with programming and the sequential model of execution presented by C++ and Python. Rather, I suggest novices are *obliged* to construct and interpret programs token-by-token and line-by-line, as they have not developed the schemata necessary to construct or interpret programs at any higher level of abstraction, such as goals or plans (Soloway & Ehrlich, 1984). This resource is productive when the semantics learners aggregate are aligned with those of the language; it is unproductive when any one interpretation of a token or line is misaligned with the language's canon.

4.1.6 *Knowledge is Authoritative*

This resource is active when learners justify the apparent semantics of tokens or lines in a program by citing their CS1 professor or an instance in which a program would work only when tokens or lines were arranged in a particular way. In a phrase, activation of this resource is characterized by learners perceiving their professor as the "sage on the stage" and programming as "getting the computer to work" (Tenenberg & Chinn, 2024). Computer science, more broadly, is conceived as memorizing and mapping rules of execution to

particular combinations of syntax and state. I hypothesize this resource is an extension of learners' general disposition to view instructors as the definitive source of knowledge, but it is reinforced in CS1 classrooms due to novices' marked unfamiliarity with programming.

4.2 RQ2: How do CS1 Novices Design, Generate, and Evaluate Programs?

Participants' conceptual resources appeared to structure how participants made sense of individual program statements, interpreted programs, and determined when programs' behavior was sufficiently explained or satisfying a problem statement. I first analyze how different resources mediated participants' design and generation of programs; I then analyze their use in program evaluation. I summarize the appearance of different resources across problems in **Table 4.2**.

Knowledge Resource	**P1**	**P2**	**P3**	**P4**	**P5**
Assignment as equality	[x]	[x]	–	–	–
Assignment as change	[x]	[x]	[x]	[x]	[x]
Assignment as transitive	–	[x]	–	[x]	–
Identifier as value	[x]	[x]	–	–	–
Knowing is incremental	[x]	[x]	[x]	[x]	[x]
Knowledge is authoritative	[x]	[x]	[x]	[x]	–

Table 4.2: Perception of participants' conceptual resources across all five interview tasks (see **Section 3.4.2**). A "[x]" indicates I perceived the resource at work in at least one student's work in my analysis; a "–" indicates I did not.

4.2.1 Program Design & Generation: Problem Decomposition

My analysis here focuses on how participants decomposed pseudocode and problem statements to design, generate, and judge the "fitness" of their code relative to problem specifications. In general, participants identified individual requirements for code in pseudocode and problem specifications. However, they viewed fulfilling those requirements as *incremental* rather than *holistic*. This resulted in programs appearing correct locally but being erroneous globally. I hypothesize these behaviors were supported by the knowledge resource *knowing as incremental* and describe relevant episodes.

S1's Decomposition of Soloway's Rainfall Problem

As the name suggests, *problem decomposition* amounts to breaking a problem up into relatively well-defined parts. Implementing a solution program then amounts to writing and stitching together subprograms that, in aggregate, solve the original problem. Resultantly, problem decomposition lessens the general cognitive effort needed design, generate, and evaluate a solution program with respect to a problem domain for programming experts and novices alike.

However, I observed novices' problem decomposition strategies were incomplete. Below is a fragment of S1's interview transcript. The fragment begins as S1 attempts to design and write a program solving the prompt listed in Problem 4 (see **Section 3.4.2**).

(1.1) **S1**: Okay, so it's asking me [to] ask the user for a number . . .
(1.2) **S1**: Okay, ask the user for [a] number, and then if the number is greater than 99,999, run a different thing. And then, average all of those inputs.
(1.3) **S1**: But, I don't know how to do that without just writing a bunch of variables asking [for an input], and then it's a fixed number of variables, and I don't want that either.
(1.4) **S1**: It there's some kind of list function [for] storing something as a list or whatever, that would be helpful. That's probably something like that out there, but I'm not sure how to implement that.

In statement 1.1, S1 immediately surmises the program should first read in an integer from standard input. As S1 continues summarizing the question prompt in statement 1.2, they list the program should check if the read-in integer is greater than 99,999 and ultimately average the sequence of read-in integers. Their use of the phrase "and then" indicates they perceive reading in, checking, and averaging as sequential steps. In statements 1.3 and 1.4, S1 acknowledges their program needs to record the sequence of read-in integers to compute the average — either as a list or sequence of discrete variables — but does not identify the syntax needed to do so. After they finish summarizing the question prompt, S1 begins writing their solution program.

S1's talk represents problem decomposition because they break the problem down into discrete steps not given by the question prompt. Implicitly, S1's summary occurs in a global `while` loop that terminates when a read-in integer is greater than 99,999, as suggested by the phrase "run a different thing" in statement 1.2. Reading a series of integers from standard input and computing their average are certainly given in the question prompt, but the steps of checking if a read-in integer is greater than 99,999, storing the integers as a list, and encapsulating these statements in a `while` loop are not.

I suggest S1's problem decomposition is "incomplete" because they initially do not consider how their enumerated steps aggregate to a program that is holistically compliant with the question prompt. For example, S1 states the program should check if a read-in integer is *greater than* 99,999, but the problem statement specifies terminating the implicit `while` loop only when a read-in integer is *equal to* 99,999. Similarly, S1 does not specify at which points the program should stop and start the loop. It is only when I begin questioning S1 about their finished program that they realize their program is incomplete.

(2.1) **I**: How did you come to think about your use of the list here?
(2.2) **S1**: Let me check if it's actually the right way.
(2.3) **S1**: Oh, is it? Maybe it should be . . .
(2.4) **S1**: No. Print the [inaudible], reading integer . . .
(2.5) **I**: What's making you —
(2.6) **S1**: Oh! It should be equals. Because it [the prompt] doesn't say more than — it

just says until this exact value [99,999] is encountered.

(2.7) **S1**: Okay, so it should be equals 99,999.
[. . .]

(2.8) **I**: How confident are you that your pseudocode meets the prompt?

(2.9) **S1**: Like 80%, 85% confident.

(2.10) **I**: What makes you think that?

(2.11) **S1**: I mean, just going and matching the natural language to each line of input.
[*S1 begins reading their program.*]

(2.12) **S1**: Does that? Oh, that's nothing here to say "ask again."

This incomplete program decomposition persists into Problem 5 (see **Section 3.4.2**). The below fragment begins after S1 claims the code listed in Problem 5 is a correct implementation of Soloway's Rainfall Problem.

(3.1) **I**: Well, how confident are you that Aubrey's program is correct?

(3.2) **S1**: Like 90% sure, I guess.

(3.3) **I**: What makes you 90% sure?

(3.4) **S1**: I don't see any syntax errors. I mean, if the plus equals what I think it means, then it looks like every variable is defined, the condition is defined, doesn't look like there's any syntax errors.
[*S1 simulates running the program to verify the absence of a divide-by-zero error.*]

(3.5) **S1**: Actually, I guess there isn't any divide by zero case, because it's going to, at least once, it's going to add your number and then add 1.

(3.6) **S1**: So then, if you just put in 99,999 right away, then it's going to print "Average: 99,999."

(3.7) **S1**: So, I guess it's fine, yeah.

In statement 3.4, S1 justifies their confidence in Aubrey's program being correct by asserting the program is syntactically valid and composed of the "parts" needed to implement a solution. In attending to whether "every variable is defined" or "the condition is defined," S1 is justifying their knowledge claim by stating the variables and conditions needed to track and aggregate a sum for an average are present. Similarly, in statement 3.5, S1 justifies their claim by stating the subprogram needed to avoid a divide-by-zero error is present. However, S1 did not recognize including the value 99,999 in the average violated the problem statement. This indicates S1's decomposition was incomplete.

S3's Decomposition of Soloway's Rainfall Problem

S3 also demonstrated application of the knowledge resource *knowing is incremental* in their design and generation of Problem 4 (see **Section 3.4.2**). The below fragment begins soon after I introduce the prompt for Problem 4 to S3. They briefly read the question prompt in silence. Their talk after their initial reading is listed below.

(4.1) **S3**: Okay. So, just to be clear, it's asking you to make a program where a user is putting in integers, and if the integer 99,999 is inputted, then it's asking to take the average of all them before 99,999, and that is when it will print the average? Just making sure that's what it's asking.
(4.2) **I**: How did you think through that?
(4.3) **S3**: Well, it says to implement a program reading integers from a standard input, so a `cin` statement, and `cin` means that a user is putting in numbers.
(4.4) **S3**: And then it says until the value of 99,999 is encountered, so once the user puts that in, it's asking it to print the average of all the integers that was put in before.

In statement 4.1, S3 attempted to confirm their decomposition of the problem presented in the question prompt. S3 decomposed the problem into four steps: (1) requesting user input; (2) checking if the input value is equal to 99,999; (3) computing the average; and (4) printing the average. At this point, S3 either did not realize a solution program would require a loop or simply did not vocalize this aspect, leaving it implicit in the causal phrase "if the integer 99,999 is inputting, then . . . " In statement 4.3, S3 appears to map the plan described in step (1) to the syntax `std::cin`. S3 reiterates their initial decomposition in statement 4.4. In the remainder of the interview, S3 often justified their knowledge by citing their professor or prior experience in their homework, indicating the activation of *knowledge is authoritative*.

After reading the question prompt, S3 attempted to implement a solution program but struggled to write syntax recording and summing the integers needed to computer the average. Before moving on to the next problem, however, I prompted S3 to describe how they thought through the code they were able to write. S3's response is in the below fragment.

(5.1) **S3**: Well, I guess I just broke the question up into parts.
(5.2) **S3**: So, I would create this as one statement, which is "Design and implement a program reading integers from standard input."
(5.3) **S3**: So, after input, that would be one part, and then until the value 99,999 is encountered, make that one part of my algorithm, and then the last part is another part which is "Print the average of the integers encountered before 99,999 to standard output," which is the output statement that I am lost on.

In statement 5.3, S3 explicitly states the problem decomposition suggested in their initial reading of the question prompt. They appear to decompose the problem into three steps: (1) receiving the user's input; (2) checking if the input is equal to 99,999, and (3) computing and printing the average.

S3 continues their decomposition strategy in Problem 5 (see **Section 3.4.2**). The below fragment begins after S3 evaluated the code listed in Problem 5 and stated it was correct. In response to their assertion, I asked them how confident they were in their assertion, and S3 responded "I'd say I'm pretty confident." I then asked them what made them so confident.

(6.1) **S3**: Because, at least, in my opinion, it looks like it's doing what the prompt is asking.
(6.2) **S3**: Because, the total is equal to the total plus the number that you put in, and if it keeps repeating that, then, until this is false, which is the if statement on line 17, if that's false, then it will then create another integer, which is the average, and then it will print out the average.

Here, their decomposition serves to assess the design of the program listed in Problem 5 with respect to the problem domain of the question prompt. In particular, the knowledge resource *knowing as incremental* serves to guide their evaluation of Aubrey's code, which is outside the scope of program design and generation, but it also guides their assessment of the program's *design* with respect to their sense of the prompt's problem domain. In statement 6.1, S3 states their confidence is supported by their perception that the program's design *looks* like it satisfies the prompt. This "looking like" amounts to, as S3 described in statement 6.2, listing the appropriate statements to record and sum a total and compute an

average. Because the program appears to satisfy the prompt in increments, S3 claimed the program is correct.

4.2.2 Program Evaluation: Manifold Conceptions of Assignment

My analysis here focuses on how participants appeared to possess many competing conceptions for the assignment operator and applied their conceptions across different contexts. All but one participant, S3, demonstrated misconceptions of assignment. In general, participants appeared to believe assignment *asserted* equality in Problem 1 but appeared to believe assignment *changed* values and, sometimes, *linked* variables with one another across time in subsequent problems. I hypothesize these interpretations were supported by the knowledge resources *assignment as equality*, *assignment as change*, and *assignment as transitive* and describe relevant episodes.

S0's Interpretations of Assignment

When asked to evaluate Problem 1 (see **Section 3.4.2**), S0 initially appeared to draw on the *assignment as equality* resource. S0 observed that the assignment operator sets the value of the variable `c` to the value of `a` but expressed confusion when reading that the program sets the value of `a` to the value of `b`. Their assertion that the statement `a = b` would be a "break in a system" if the values of `a` and `b` were not equal suggests they interpreted the assignment operator as an *assertion of equality*, similar to the equals sign in mathematics.

(7.1) **S0**: So, therefore, whatever integer I had put in for integer `a` at the very beginning is going to also be duplicated for the integer `c`.
(7.2) **S0**: And then, there's another statement, I'm assuming, with `a` equaling to `b`, but [the reason] that confuses me is because it said [that] after we have already inputted a number for integer `b` right [. . .]
(7.3) **S0**: That looks like a break, almost, like a break in a system, especially if they're not equal.

When I asked S0 to elaborate on the nature of the error, S0 began explaining that if `a` and `b` were to equal 10 and 12 respectively, the computer would interpret the statement `a = b`

as `10 = 12` and resultantly report an error. However, S0 immediately raised the possibility that the computer would simply accept the assertion that 10 is equal to 12.

(8.1) **S0**: It'd tell me, "Hey, I'm looking at different statements here, and there aren't adding up. First, I implemented two `cin`s and with their two different numbers, and now you're telling me `a` equals to `b`, but 10 does not equal to 12."
(8.2) **S0**: Maybe it would work, and the system would assume that 10 equals 12, and wouldn't put an actual . . . essential element behind a number.
(8.3) **S0**: Rather, that they're just going to follow blindly and says `a` equals to `b`, and it may just go with it.
(8.4) **S0**: Or, the computer may look at that mathematical value as an essential component and say, "No, look for 10, I have 10 bytes," for example, I'm just throwing my thoughts out there [. . .] "10 bytes and 12 bytes don't equal for me."

Note that S0 expressed no confusion at the assignment `c = a`. I hypothesize *assignment as equality* is an easily cued knowledge resource and that its explanatory power in interpreting the assignment `c = a` cemented its activation throughout S0's evaluation of Problem 1.

When evaluating Problem 2 (see **Section 3.4.2**), S0 read out that the variables `x` and `number` are assigned the value of 0. I hypothesize that during this initial read-out, the resource *assignment as equality* is active. They appear to briefly have trouble interpreting the condition `x < number` and, resultantly, quickly abandon *assignment as equality* in favor of *assignment as mutation* to explain the condition, taking note that the value of `number` is changed by a `cin` statement earlier in the program.

(9.1) **S0**: So, we started with including the `iostream` for the computer to work with us, and then we introduced the `int main` [. . .] and introduced the variable `x` right after in line four.
(9.2) **S0**: And then, line six, we've introduced the `number` [varriable] [. . .] and we've equaled it to zero.
(9.3) **S0**: If `x` is less than number . . . If `x` is less than number.
(9.4) **S0**: Okay. So `x` is less than number, right?
(9.5) **S0**: But I just got caught up because we introduced the integer `x` in the very beginning, equalling to zero.
(9.6) **S0**: So, `number` and `x` are already equal, they're both zero, so . . . This doesn't make sense because . . .

(9.7) **S0**: Nope, we started the program with the variables' value being zero, but that does not mean it's permanently changed for the entire code. We can change that along the way.
(9.8) **S0**: So, when we started with `std::cin << number` in code seven, we've just implemented a new value for the variable `number`. So that's no problem.

S0 continued explaining the program's behavior. To explain the program's behavior in the `while` loop, S0 began simulating the program's execution with input values 3 and 4 and claimed the program would print the value 4 at line 16. At this point, S0 appeared to neglect the fact that the `while` construct will cause the computer's instruction pointer to jump from line 14 to line 8 after executing `std::cin >> number`, leading to their mistaken interpretation that the program will immediately print `x` after reading in a value for `number`. Despite this mistake, it is clear S0 is drawing on the resource *assignment as transitive* to evaluate lines 13 and 16, as the value of `x` at line 16 is set to the value fed into `number` at line 13.

(10.1) **S0**: And then it's going to change the `x` value into equaling that number.
(10.2) **S0**: So, if we were to put 3, for example, this value [`number`] is going to be 3.
(10.3) **S0**: So, now `x` is going to equal to 3, because `x` being 0 is less than 3.
(10.4) **S0**: And then we're going to move down into having to implement another value for the variable `number`.
(10.5) **S0**: So, we could say 4 here [. . .]
(10.6) **S0**: And, in line 16, it's going to show that `x` equals 4.
(10.7) **S0**: And that happened because `x` is already equal to `number` [. . .]
(10.8) **S0**: So, [for] whatever new integer we implement into the variable `number`, it is going to be [implemented] into the variable `x`.

I hypothesize that the knowledge resource *assignment as equality* deactivated in response to its lack of explanatory power for Problem 2. The resources *assignment as mutation* and *assignment as transitive* activated to construct an explanation of the program's behavior. As a result, S0 interpreted the assignment operator as changing the value of the variable on its left-hand side — an interpretation closely aligned with normative interpretations of the operator — but additionally interpreted the operator as establishing a relation-

ship between the operands. This led to him mistakenly claiming that the value of `x` would take on whatever value `number` is set to.

S5's Interpretations of Assignment

When asked to evaluate Problem 1 (see **Section 3.4.2**), S5 began by describing the program's behavior line-by-line. After reading the assignment statements for variables `a`, `b`, and `c`, S5 quickly concluded the program is setting the values of `a`, `b`, and `c` equal to one another.

(11.1) **S5**: I know this [`#include <iostream>`] is up here because it gives you access to input and output commands, which are down here.
(11.2) **S5**: This [`int a; int b;`] is storing these as integer values.
(11.3) **S5**: These [`std::cin`] are both character input commands with, I don't remember what that's called [presumably referring to `>>`].
(11.4) **S5**: And it's giving this as an input as `a`, this is an input as `b`, and then down here it's saying that `c` is equal to `a`, `a` is equal to `b`, and `b` is equal to `c`, so I guess it's saying they're all equal.
(11.5) **S5**: And then it's ending the program.

To better understand S5's evaluation, I asked them how the program would behave if the user submitted values 2 and 5 when the program asked for user input. S5 responded that the program would "tell you that `c` is equal to 2, 2 is equal to 7, and 7 is equal to 2." I asked S5 what the program's supposed statement of `7 = 2` might mean. S5 responded, "I don't know, because seven is not equal to two." A few minutes later, I asked S5 how confident they were in their interpretation of the program's behavior.

(12.1) **I**: How confident are you in you answer?
(12.2) **S5**: Pretty confident.
(12.3) **I**: What makes you confident?
(12.4) **S5**: I don't know. It seems pretty straightforward.
(12.5) **S5**: That if you're inputting `a` and `b` and they're equal to each other, that `c` is `b`, which is equal to `a`.

S5's statements suggest the resource *identifier as value* is active during their evaluation of Problem 1. S5 successfully interpreted the include, declaration, and standard input

statements leading up to the swap, but they mistakenly interpreted the swap as setting the variables as equal to one another. That this results from activation of *identifier as value* is suggested by their interpreting of the program as stating `7 = 2`. The identifiers `a` and `b` are replaced by the values `7` and `2`. I distinguish this resource from *assignment as equality* because, unlike S0, S5 did not seem to believe the statement would cause an error.

I conclude that the resource *identifier as value* is activated throughout S5's interpretation of Problem 1. Additionally, I suggest that this resource remains activated because it supports a satisfying narrative for a subset of possible inputs to Problem 1. In S5's final statement, they imply the program is straightforward for cases where the user submits equal values of `a` and `b`. In these cases, *identifier as value* is productive. Why S5 did not find it sufficiently problematic that the program appears to state propositions like `7 = 2` for other cases is not clear.

When asked to evaluate Problem 2 (see **Section 3.4.2**), S5 began by describing the program's behavior line-by-line. However, as they begin describing the behavior of the statement `x = number` in relation to the condition `x < number`, S5 doubted their understanding of the assignment operator.

(13.1) **S5**: While it [`number`] doesn't equal negative one, then it [`x`] equals less than zero [in reference to `x < number`].
(13.2) **S5**: But then it says zero equals `number`, and I'm back to my same problem where I don't know what the equals sign means.

Note that S5 omitted the statement `std::cin >> number` and misinterpreted the semantics of the statement `if (x < number)` in this episode. Hence, they appeared to believe the statement `(x < number)` sets the value of `x` to be less than `number`, which they believed is still zero. Despite this, S5 continued describing the program's behavior. S5 concluded their description by stating the program will print the integer most recently submitted to the program.

(14.1) **S5**: So, the [variable] `number` does not equal negative one, yeah? Because the [variable] `number` is zero.
(14.2) **S5**: So, while the [variable] `number` does not equal negative one, `x`, which is zero, is less than the [variable] `number`.
(14.3) **S5**: `x` equals `number`.
(14.4) **S5**: And then, you're going to input another number, and it's just gonna print that number back out to you.

Note that S5 misinterpreted the semantics of the `while` construct during this episode. Earlier, they remarked that the loop would repeat at least once, but it is unknown why they appear to arbitrarily terminate the loop during their explanation here.

S5 readily describing the program's behavior line-by-line suggests that they believe, or act as if, knowledge of a program is constructed line-by-line. This contrasts with the naturalistic epistemology of experts, who tend to construct interpretations of programs as if knowledge were built by recognition and composition of multi-line patterns of code. From this, I conclude that the resource *knowledge is incremental* was active in S5's conceptual ecology during their interpretation. Additionally, S5's interpreting of the statement `std::cout << x` as printing the value *most recently* submitted to the variable `number` indicates they believe the assignment operator "links" the values of its two operands. That is, they believe mutating `number` mutates `x`. From this, I conclude S5 used the resource *assignment as transitive* to evaluate the program.

S5's change from *identifiers as values* to *assignment as transitive* suggests some feature of the problem context deactivated and activated these resources, respectively. I hypothesize this occurred because the resource *identifier as value* was not productive when interpreting assignment operations across time, as required by the existence of the `while` operator. Repeated executions of the statement `x = number` coupled with the initial initialization of `x` to zero and printing of `x` suggested to S5 that `x` had to change. Thus, *assignment as mutation* activated to accommodate change, with *assignment as transitive* following closely.

4.3 Summary

In this chapter, I first described the knowledge resources novice computer science students possess as they enter CS1 classrooms. I then described how these resources structured novices' practice in program design, generation, and evaluation. In particular, I found that novices enter CS1 classrooms with manifold conceptions of the assignment operator and dispositions for how knowledge is constructed in program design, generation, and evaluation. Participants' conceptions of assignment most prominently affected how they evaluated programs, but which conceptions they used varied problem-by-problem. This suggests contextual features of the problems influenced the activation of different assignment resources. Participants' dispositions for how knowledge is constructed most prominently affected how participants satisfied themselves with their explanations of how a program's design, syntax, or behavior met a specified design, pseudocode, or question prompt.

Two knowledge resources were constant across participants' design, generation, and evaluation of programs. *Assignment as change* was observed in all problems, though its observation in Problem 1 is credited to one participant, S1, demonstrating a canonical interpretation of the assignment operator. However, this suggests participants largely understood the assignment operator as changing values but features of Problem 1 constrained their interpretation such that *assignment as equality* dominated. The second constant resource was *knowing is incremental*. I hypothesize its constant presence is owed to the fact that participants are *novices*, so they are obligated to construct and evaluate programs token-by-token and line-by-line. The success of this strategy was contingent on their knowledge of the semantics of individual tokens or lines.

CHAPTER V: Conclusion

In this thesis, I demonstrated computer programming novices possess a diversity of conceptual resources at the start of a Computer Science I course. In particular, I showed the following.

1. Computer science novices enter Computer Science I courses with conceptions for knowledge in and about programming and many, competing conceptions about assignment.

2. Computer science novices apply these conceptions to incrementally build knowledge claims about programs but fail to construct holistic summaries of programs and problem statements with appropriate semantics for syntax elements.

The implications of these findings are as follows but expanded on below. First, the fact that novices enter computer science with intuitions about when an explanation of a program's behavior is satisfactory suggests a target for instruction. If instructors can impress on novices expert-level criteria for explanations of programs, novices may more effectively engage in their learning by seeking resources or justifications for appropriate explanations. Second, the existence of many conceptual resources about the assignment operator implies a similar suite of resources for other syntactical elements, which may similarly be targets for instruction.

5.1 Discussion

Learning to program is, of course, notoriously difficult. Yet, programming is rapidly becoming an in-demand skill for not only economic purposes but also for effectively participating in the democratic life of an increasingly technological world. As a result, numerous theories and models of programming as an *activity* and a *thing* to be learned have been proposed to

ease computing instructors and learners' difficulties. In particular, computer science education researchers have proposed learning to programming generates *misconceptions* that, if left uncorrected, complicate the learning process (Qian & Lehman, 2017).

However, misconceptions-focused research risks obscuring the competencies of novice programmers and provide little traction for design-based computing education researchers (Danielak, 2019; Nelson & Ko, 2018). Further, computing education researchers have increasingly argued for domain-specific theories of learning and cognition in computer science (Szabo & Sheard, 2022; Tedre & Pajunen, 2022). This thesis builds on these ideas by asking: What would would a computing-specific theory of novices' programming knowledge and practice look like? In surfacing a model of computer science novices' intuitions *about* knowledge of and about programs and varying conceptions of the assignment operator, I believe this thesis provides a preliminary answer.

First, such a theory can objectify the kinds of knowledge used and required to program and track their use in learners' programming practice. This is an extension of prior work attempting to synthesize the kind of knowledge needed to program (Robins et al., 2003), but it also raises the possibility of extending more precise, discrete theories of programming knowledge. For example, Nelson (2021) observed that all compilers and interpreters for programming languages encode exactly the kind of knowledge needed to evaluate programs: *rules of execution* and *mappings* between a programming language's syntax, semantics, and a program's state. This theory of knowledge enabled Nelson to systematically develop instruction and assessments that taught and tested learners at a granular level. However, Nelson conceptualized this knowledge in terms of how it might appear among experts, so it is not clear how such knowledge might be conceived or used by novices — nor how novice-level knowledge might change into expert-level knowledge. This thesis, in identifying novices' various conceptions of assignment and tracking their use in program evaluation, suggests that this knowledge works to explain program behavior locally or line-by-line rather than as a coherent whole. Additionally, as learners' conceptions of

assignment may constitute "rules of execution," this thesis may serve as the basis of future work seeking to form a continuity between novices' rules of execution and those observed in the compilers and interpreters of programming languages.

Second, such a theory can admit a diversity of knowledge elements in its descriptions of novice programming practice and thereby recast misconceptions as potentially productive conceptions. Again, this is an extension of prior work attempting to identify conceptual resources among programming novices (Chao, 2012; Lewis, 2012), but little emphasis has been given to how such diversity can account for novices' misconceptions. In this thesis, I gave particular emphasis to how resources could lead to productive and unproductive emergent models of programs and programming languages. This emphasis opens the door to how an environment's *conceptual dynamics* may influence novices' dynamic conceptions of disciplinary content (Sherin et al., 2012). Chao et al. (2018) has explored these dynamics in the context of program recursion and assessments, but this thesis sheds insight into how these dynamics might be conceptualized for general program design, generation, and evaluation. Thus, this thesis may serve as a basis for further inquiry regarding how problem contexts affect novices' construction of productive conceptions or misconceptions, informing instructional design while sustaining their existing competencies.

Finally, such a theory can provide *targets for instruction*. If aligned with the kind of systematic, granular assessment performed by Nelson (2021), a resource-based theory of computing knowledge and cognition enables instruction to target the activation and coordination of such resources. Based on this thesis' findings, "correcting" students' misconceptions must amount to more than telling them they are wrong and detailing the correct conception. Correction must acknowledge that students' conceptions are perceived as useful and target these conceptions within that perspective. Further, this work may serve as the basis for design-based research of instruction. If students' construction of assignment is partly determined by the programs they are reading, what would it look like to structure a learning module acknowledging this? How might learning occur if the example programs

novices encountered were carefully curated to support only canonical constructions of assignment? Future work marshalling the resources discussed here in concert with similar resource-based work in the computer education research community may shed insight on these questions.

5.2 Limitations

A central limitation of this study is quality and quantity of the interviews. Though six interviews has been named the minimum for data saturation in qualitative research (Guest et al., 2006), it is likely additional interviews would have provided a greater diversity of student thinking. Further, five out of six participants were recruited from the same class. Greater variation in and size of the participant population would have demanded greater cohesion across my analysis, strengthening the applicability and extendability of my results. In addition, though researchers in the Knowledge in Pieces tradition have argued for the ecologically validity of clinical interviews (diSessa, 2007), it is likely presenting participants interview tasks *without* access to a text editor and compiler or interpreter prevented them from deploying resources they ordinarily would in their practice (Litherland & Kluge, 2023). Thus, this study neglects a crucial dimension of novice programmers' knowledge and strategies.

This study is also limited by the duration and quality of my analysis. Qualitative analyses situated in Knowledge in Pieces or resource theories of cognition and learning often require either extending extant theories or building new theory. Generating or extending theory that is both accountable to the data and described well enough to be extended by others requires time. In particular, it requires prolonged immersion in, negotiation with, and rigorous interpretation of the data (Parnafes & Disessa, 2013). The conceptual resources presented here may require additional development, as their current formulations may change given time not permitted by the timeline of this thesis project.

Finally, this study is limited by lack of insight into participants' classrooms. Knowledge of participants' lectures, closed lab assignments, and homework would have revealed the contexts in which participants learned about program execution, assignment, conditionals, and control structures. This contextual knowledge could have supported inferences as to why participants possessed and deployed the conceptual resources they did, strengthening the educational implications of this work.

5.3 Future Directions

The limitations listed above immediately raise opportunities for future work. Such work should seek a greater diversity of novice computer science undergraduates for the purpose of building and testing theories about novice learning and cognition. Further, as such theories develop, future work should seek to integrate them with accounts of computer science undergraduates' cognition in higher level courses, such as those involving compiler design or software development. Additionally, future work should seek to extend extant learning theory within and without the computer science education community; if extension is deemed inappropriate, future work should discuss why their theory is better suited. Finally, future work should relate their analysis to the happenings of participants' classrooms.

There are also broader areas for future work. For example, greater insight into novices' conceptual change in regards to the assignment operator may be gleaned if clinical interviews were repeated throughout the semester. If paired with an analysis of the happenings of the classroom, this work may inform our understanding of the conceptual dynamics of the classroom in contrast with interview settings. Such an understanding would not only inform instructors' understandings of novices' concepts and cognition but also researchers' understandings as they analyze interviews.

Future work may also explore the relationship between novices' conceptions and the programming language with which they are learning computer science. Here, I do not note the differences between C++ and Python, but programming languages present different syn-

tax and semantics for the same symbols and, ultimately, present contrasting models of the computer to programmers (Fincher et al., 2020). Prior work has identified that novices possess conceptions of how solutions might be expressed in programming languages (Pane et al., 2001), and other work has posited misconceptions may emerge from a mismatch between these conceptions and the tools presented by a programming language (Soloway et al., 1983). A program of research uncovering the relations between novices' conceptions and the introductory programming language may support future research into the design of educational programming languages. In total, such a program would support the *re-structuration* of introductory computer science, as a programming language designed for novice' conceptions would not merely be teaching computer science *better* but changing *what* is taught to novices (Wilensky & Papert, 2010).

References

Ben-Ari, M. (2001). Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, *20*(1), 45–73. https://doi.org/10.1145/274790.274308

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32–36. https://doi.org/10.1145/1272848.1272879

Blikstein, P., & Moghadam, S. H. (2019). Computing Education: Literature Review and Voices from the Field. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge Handbook of Computing Education Research* (pp. 56–78). Cambridge University Press.

Bonar, J., & Soloway, E. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human–Computer Interaction*, *1*(2), 133–161. https://doi.org/10.1207/s15327051hci0102_3

Brown, N. J. S., A. Danish, J., Levin, M., & A. diSessa, a. A. (2015). Competence Reconceived: The Shared Enterprise of Knowledge Analysis and Interaction Analysis. In *Knowledge and Interaction*. Routledge.

Chao, J. (2012). *Phenomenological Primitives in Introductory Computer Science Students' Understanding of Recursion* [Doctoral dissertation, University of Virginia]. Retrieved February 25, 2024, from https://doi.org/10.18130/V3JV7J

Chao, J., Feldon, D. F., & Cohoon, J. P. (2018). Dynamic Mental Model Construction: A Knowledge in Pieces-Based Explanation for Computing Students' Erratic Performance on Recursion. *Journal of the Learning Sciences*, *27*(3), 431–473. https://doi.org/10.1080/10508406.2017.1392309

Clancy, M. J. (2004). Misconceptions and Attitudes that Interfere with Learning to Program. In *Computer Science Education Research*. Taylor & Francis.

Clement, J. (2000). Analysis of Clinical Interviews: Foundations and Model Viability. In *Handbook of Research Design in Mathematics and Science Education*. Routledge.

Cohen, L., Lawrence, M., & Keith, M. (2017). Coding and content analysis. In *Research Methods in Education* (8th ed.). Routledge.

Danielak, B. A. (2019). Deprecating Misconceptions through Context-Dependent Accounts of Productive Knowledge. *Proceedings of the 2019 ACM Conference on International Computing Education Research*, 91–100. https://doi.org/10.1145/3291279.3339424

Danielak, B. A. (2022). How Code Takes Shape: Studying a Student's Program Evolution. *Cognition and Instruction*, *40*(2), 266–303. https://doi.org/10.1080/07370008.2022.2044330

Dehnadi, S., & Bornat, R. (2006). *The Camel Has Two Humps (working title)*. Retrieved January 6, 2024, from https://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

diSessa, A. A. (1993). Toward an Epistemology of Physics. *Cognition and Instruction*, *10*(2-3), 105–225. https://doi.org/10.1080/07370008.1985.9649008

diSessa, A. A. (2006). A History of Conceptual Change Research: Threads and Fault Lines. In *The Cambridge handbook of: The learning sciences* (pp. 265–281). Cambridge University Press.

diSessa, A. A. (2007). An Interactional Analysis of Clinical Interviewing. *Cognition and Instruction*, *25*(4), 523–565. https://doi.org/10.1080/07370000701632413

diSessa, A. A., L. Sherin, B., & Mariana Levin, a. (2015). Knowledge Analysis: An Introduction. In *Knowledge and Interaction*. Routledge.

diSessa, A. A., & Sherin, B. L. (1998). What changes in conceptual change? *International Journal of Science Education*, *20*(10), 1155–1191. https://doi.org/10.1080/0950069980201002

Doroudi, S. (2021, August 11). *A Primer on Learning Theories*. https://doi.org/10.35542/osf.io/ze5hc

du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Duit, R., & Treagust, D. F. (2003). Conceptual change: A powerful framework for improving science teaching and learning. *International Journal of Science Education*, *25*(6), 671–688. https://doi.org/10.1080/09500690305016

Fereday, J., & Muir-Cochrane, E. (2006). Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development. *International Journal of Qualitative Methods*, *5*(1), 80–92. https://doi.org/10.1177/160940690600500107

Fincher, S., Jeuring, J., Miller, C. S., Donaldson, P., du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J. L., & Petersen, A. (2020). Notional Machines in Computing Education: The Education of Attention. *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 21–50. https://doi.org/10.1145/3437800.3439202

Gaddis, T. (2018). *Starting out with C++. From control structures through objects* (Ninth edition). Pearson.
OCLC: 965781245.

Gaddis, T. (2020). *Starting out with Python* (Fifth edition). Pearson.

Guest, G., Bunce, A., & Johnson, L. (2006). How Many Interviews Are Enough?: An Experiment with Data Saturation and Variability. *Field Methods*, *18*(1), 59–82. https://doi.org/10.1177/1525822X05279903

Hammer, D., & Berland, L. K. (2014). Confusing Claims for Data: A Critique of Common Practices for Presenting Qualitative Research on Learning. *Journal of the Learning Sciences*, *23*(1), 37–46. https://doi.org/10.1080/10508406.2013.802652

Hammer, D., & Elby, A. (2003). Tapping Epistemological Resources for Learning Physics. *Journal of the Learning Sciences*, *12*(1), 53–90. https://doi.org/10.1207/S15327809JLS1201_3

Herman, G. L., Loui, M. C., Kaczmarczyk, L., & Zilles, C. (2012). Describing the What and Why of Students' Difficulties in Boolean Logic. *ACM Trans. Comput. Educ.*, *12*(1), 3:1–3:28. https://doi.org/10.1145/2133797.2133800

Huang, Y., Schunn, C. D., Guerra, J., & Brusilovsky, P. (2024). Why Students Cannot Easily Integrate Component Skills: An Investigation of the Composition Effect in Programming. *ACM Trans. Comput. Educ.* https://doi.org/10.1145/3673239
Just Accepted.

Ko, A. J., Myers, B. A., & Aung, H. H. (2004). Six Learning Barriers in End-User Programming Systems. *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 199–206. https://doi.org/10.1109/VLHCC.2004.47

Ko, A. J., Oleson, A., Ryan, N., Register, Y., Xie, B., Tari, M., Davidson, M., Druga, S., & Loksa, D. (2020). It is time for more critical CS education. *Communications of the ACM*, *63*(11), 31–33. https://doi.org/10.1145/3424000

Levin, D. T., Hammer, D., Elby, A., & Coffey, J. E. (2013). The Refinement of Everyday Thinking. In *Becoming a responsive science teacher: Focusing on student thinking in secondary science* (Online-Ausg, pp. 15–42). NSTA Press/National Science Teachers Association.

Lewis, C. M., Clancy, M. J., & Vahrenhold, J. (2019). Student Knowledge and Misconceptions. In A. V. Robins & S. A. Fincher (Eds.), *The Cambridge Handbook of Computing Education Research* (pp. 773–800). Cambridge University Press. Retrieved May 21, 2024, from https://doi.org/10.1017/9781108654555.028

Lewis, C. M. (2012). *Applications of out-of-domain knowledge in students' reasoning about computer program state* [Doctoral dissertation, University of California at Berkeley]. https://dl.acm.org/doi/book/10.5555/2518756

Litherland, K., & Kluge, A. (2023). Learning to program as empirical inquiry: Using a conversation perspective to explore student programming processes. *Computer Science Education*, 1–25. https://doi.org/10.1080/08993408.2023.2290410

Luxton-Reilly, A. (2016). Learning to Program is Easy. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 284–289. https://doi.org/10.1145/2899415.2899432

McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K., & Zander, C. (2017). Folk Pedagogy and the Geek Gene: Geekiness Quotient. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 405–410. https://doi.org/10.1145/3017680.3017746

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

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, 125–180. https://doi.org/10.1145/572133.572137

Miedema, D., Aivaloglou, E., & Fletcher, G. (2021). Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study. *Proceedings of the 17th ACM Conference on International Computing Education Research*, 355–367. https://doi.org/10.1145/3446871.3469759

Miller, C. S., & Settle, A. (2019). Learning to Get Literal: Investigating Reference-Point Difficulties in Novice Programming. *ACM Trans. Comput. Educ.*, *19*(3), 28:1–28:17. https://doi.org/10.1145/3313291

Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies*, *6*(2), 237–260. https://doi.org/10.1016/S0020-7373(74)80004-0

Nelson, G. L., & Ko, A. J. (2018). On Use of Theory in Computing Education Research. *Proceedings of the 2018 ACM Conference on International Computing Education Research*, 31–39. https://doi.org/10.1145/3230977.3230992

Nelson, G. L. (2021, August 26). *Teaching and Assessing Programming Language Tracing*. Retrieved October 28, 2024, from http://hdl.handle.net/1773/47433

Noonan, R. (2017). *STEM Jobs: 2017 Update. ESA Issue Brief #02-17*. US Department of Commerce. Retrieved April 6, 2024, from https://eric.ed.gov/?id=ED594354

Oliveira, E., Keuning, H., & Jeuring, J. (2023). Student Code Refactoring Misconceptions. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 19–25. https://doi.org/10.1145/3587102.3588840

Pane, J. F., Ratanamahatana, C., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*(2), 237–264. https://doi.org/10.1006/ijhc.2000.0410

Parnafes, O., & Disessa, A. A. (2013). Microgenetic Learning Analysis: A Methodology for Studying Knowledge in Transition. *Human Development*, *56*(1), 5–37. https://doi.org/10.1159/000342945

Pea, R. D. (1986). Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, 204–223. https://doi.org/10.1145/1345443.1345441

Pintrich, P. R., Marx, R. W., & Boyle, R. A. (1993). Beyond Cold Conceptual Change: The Role of Motivational Beliefs and Classroom Contextual Factors in the Process of Conceptual Change. *Review of Educational Research*, *63*(2), 167–199. https://doi.org/10.3102/00346543063002167

Posner, G. J., Strike, K. A., Hewson, P. W., & Gertzog, W. A. (1982). Accommodation of a scientific conception: Toward a theory of conceptual change. *Science Education*, *66*(2), 211–227. https://doi.org/10.1002/sce.3730660207

Qian, Y., & Lehman, J. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education*, *18*(1), 1–24. https://doi.org/10.1145/3077618
00100.

Robins, A. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, *20*(1), 37–71. https://doi.org/10.1080/08993401003612167

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Sherin, B. L. (2001). How Students Understand Physics Equations. *Cognition and Instruction*, *19*(4), 479–541. https://doi.org/10.1207/S1532690XCI1904_3

Sherin, B. L., Krakowski, M., & Lee, V. R. (2012). Some assembly required: How scientific explanations are constructed during clinical interviews. *Journal of Research in Science Teaching*, *49*(2), 166–198. https://doi.org/10.1002/tea.20455

Smith III, J. P., diSessa, A. A., & Roschelle, J. (1994). Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *Journal of the Learning Sciences*, *3*(2), 115–163. https://doi.org/10.1207/s15327809jls0302_1

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858. https://doi.org/10.1145/6592.6594

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, *26*(11), 853–860. https://doi.org/10.1145/182.358436

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609. https://doi.org/10.1109/TSE.1984.5010283

Sweller, J. (1988). Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, *12*(2), 257–285. https://doi.org/10.1207/s15516709cog1202_4

Swidan, A., Hermans, F., & Smit, M. (2018). Programming Misconceptions for School Students. *Proceedings of the 2018 ACM Conference on International Computing Education Research - ICER '18*, 151–159. https://doi.org/10.1145/3230977.323099500009.

Szabo, C., & Sheard, J. (2022). Learning Theories Use and Relationships in Computing Education Research. *ACM Trans. Comput. Educ.*, *23*(1), 5:1–5:34. https://doi.org/10.1145/3487056

Taylor, J. (1990). Analysing novices analysing Prolog: What stories do novices tell themselves about Prolog? *Instructional Science*, *19*(4), 283–309. https://doi.org/10.1007/BF00116442

Tedre, M., & Pajunen, J. (2022). Grand Theories or Design Guidelines? Perspectives on the Role of Theory in Computing Education Research. *ACM Trans. Comput. Educ.*, *23*(1), 4:1–4:20. https://doi.org/10.1145/3487049

Tenenberg, J. (2019). Qualitative Methods for Computing Education. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge Handbook of Computing Education Research* (pp. 173–207). Cambridge University Press.

Tenenberg, J., & Chinn, D. (2024). Epistemic practices in conceptions of computer science. *Computer Science Education*, 1–22. https://doi.org/10.1080/08993408.2024.2381398

U.S. Department of Labor. (2007). The STEM Workforce Challenge: The Role of the Public Workforce System in a National Solution for a Competitive Science, Technology, Engineering, and Mathematics (STEM) Workforce. Retrieved April 6, 2024, from https://hdl.handle.net/1813/78444

Vivian, R., Falkner, K., & Szabo, C. (2014). Can everybody learn to code?: Computer science community perceptions about learning the fundamentals of programming. *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, 41–50. https://doi.org/10.1145/2674683.2674695

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 39–44. https://doi.org/10.1145/2591708.2591749

Wilensky, U., & Papert, S. (2010). Restructurations: Reformulating Knowledge Disciplines through New Representational Forms.

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., & Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education, 29*(2-3), 205–253. https://doi.org/10.1080/08993408.2019.1565235

Xinogalos, S. (2015). Object-Oriented Design and Programming: An Investigation of Novices' Conceptions on Objects and Classes. *ACM Trans. Comput. Educ., 15*(3), 13:1–13:21. https://doi.org/10.1145/2700519

Appendices

APPENDIX A: Recruitment Materials

1.1 Survey

About You

1. **Your name:** _____
2. **How old are you?** _____
3. **What is your major(s)?** _____
4. **What is your minor(s)?** _____
5. **Contact Information** *(Please list an email or phone number you check regularly. This will be used to contact you when scheduling the interview.)*

   _____

Programming Experience

4. **How did you first learn to program? Please select only one option.**

   (a) □ In a class in or before high-school.

   (b) □ At a job.

   (c) □ Self-taught before my Computer Science course(s).

   (d) □ Right now, in my Computer Science course(s).

   (e) □ Other (please describe) _____

5. **What programs have you written outside of your Computer Science courses(s)? Describe what they do in detail. Write N/A if you have not written any programs outside of your Computer Science course(s).**

6. **Are you proficient at writing programs in the Python programming language? Please select only one option.**

□	□	□	□
Not at all	Somewhat	Slightly	Very

7. **Are you proficient at writing programs in the C++ programming language? Please select only one option.**

□	□	□	□
None at all	Somewhat	Slightly	Very

8. **Aside from Python and C++, what other programming languages are you proficient in? Indicate your proficiency as either "Not at all," "Somewhat," "Slightly," or "Very."**

1.2 Screening Rubric

	Inclusion	**Exclusion**
Question 4	Selects either (c) or (d), or answer to (e) indicates no or minimal programming experience.	Selects either (a) or (b), or answer to (e) indicates extensive or formal programming experience.
Question 5	Leaves answer space blank or writes exactly or variant of "N/A". Or description of programming project(s) includes only CS1-level terminology, data structures, or algorithms, such as "function", "list", or "find the minimum". And nature of described programming project(s) requires the use of only CS1-level computer science concepts, such as program arithmetic, conditionals, printing, or iteration.	Description of programming project(s) includes computer science terminology, data structures, or algorithms reserved for post-CS1 computer science courses, such as "selection sort", "stacks", "concurrent", or "relational database". Or nature of described programming project(s) requires the use of post-CS1 computer science concepts, such as bit-string manipulation, concurrency, relational databases, or parsing.
Question 6	Selects either "None at all" or "Somewhat".	Selects either "Slightly" or "Very".
Question 7	Selects either "None at all" or "Somewhat".	Selects either "Slightly" or "Very".
Question 8	Either lists no languages or description of proficiency in any listed language is either "None at all" or "Somewhat".	Description of proficiency in any listed language is either "Slightly" or "Very".

Table A.1: A rubric to determine whether a survey participant may be invited to participate in a compensated task-based interview. A participant is included when their answer to survey questions 4, 5, 6, 7, and 8 fulfill all criteria under the "Inclusion" column. A participant is excluded when their answer to survey questions 4, 5, 6, 7, and 8 fulfill one or more of the criteria under the "Exclusion" column. If the description of the programming project(s) indicate language proficiency lesser or greater than the language proficiencies listed in questions 6, 7, or 8, participants may be included or excluded accordingly.

1.3 Sorting Rubric

	More Indicative of Novice	Less Indicative of Novice
Question 4	Selects (d) or answer to (e) indicates no programming experience.	Selects either (a), (b), or (c), or answer to (e) indicates minimal, extensive, or formal programming experience.
Question 5	Leaves answer space blank or writes exactly or variant of "N/A".	Description of programming project(s) includes computer science terminology, data structures, or algorithms reserved for CS1 or post-CS1 computer science courses, such as "function", "list", "recursion", or "concurrency". Or nature of described programming project(s) requires the use of CS1 or post-CS1 computer science concepts, such as program arithmetic, iteration, bit-string manipulation, or parsing.
Question 6	Selects "None at all".	Selects either "Somewhat", "Slightly", or "Very".
Question 7	Selects "None at all".	Selects either "Somewhat", "Slightly" or "Very".
Question 8	Leaves answer space blank, does not list any languages, or description of proficiency in any listed language is "None at all".	Description of proficiency in any listed language is either "Somewhat", "Slightly" or "Very".

Table A.2: A rubric to determine the order in which survey participants may be contacted for invitation to participate in a compensated task-based interview when there are more than 6 qualified survey participants. Participants whose answers to survey questions 4, 5, 6, 7, and 8 most meet the criteria under the "More Indicative of Novice" column will be contacted first. For example, a participant whose answers to survey questions 4, 5, and 8 match the criteria for "More Indicative of Novice" but 6 and 7 match the criteria for "Less Indicative of Novice" will be contacted before a participant whose answers to survey questions 5 match the criteria for "More Indicative of Novice" but 4, 6, 7, and 8 match the criteria for "Less Indicative of Novice".

APPENDIX B: Interview Tasks

2.1 C++ Tasks

1. Consider the below program. What does the program do?

```
#include <iostream>

int main() {
    int a;
    int b;

    std::cin >> a;
    std::cin >> b;

    int c = a;
    a = b;
    b = c;

    return 0;
}
```

2. Consider the below program. What does the program do?

```
#include <iostream>

int main() {
    int x = 0;

    int number = 0;
    std::cin >> number;
    while (number != -1) {
        if (x < number) {
            x = number;
        }

        std::cin >> number;
    }

    std::cout << x;

    return 0;
}
```

3. Implement the below algorithm as a program.

```
read in a number N

initialize number P
set P equal to 1

initialize number J
set J equal to 1

repeat the following until J is greater than N:
    set P equal to P multiplied by J
    increment J by 1

print P
```

4. Design and implement a program reading integers from standard input until the value 99,999 is encountered. Print the average of the integers encountered before 99,999 to standard output. Ensure your program works for all inputs.

5. Aubrey, a CSCI 1170 student, was asked to design and implement a program according to the following prompt.

   > Design and implement a program reading integers from standard input until the value 99,999 is encountered. Print the average of the integers encountered before 99,999 to standard output. Ensure your program works for all inputs.

   Below is the program Aubrey designed and implemented. Why is Aubrey's program correct/incorrect?

```
1  #include <iostream>
2
3  const int SENTINEL = 99999;
4
5  int main() {
6      int total = 0;
7      int count = 0;
8
9      int number = 0;
10     bool no_sentinel = true;
11
12     std::cin >> number;
13     while (no_sentinel) {
14         total += number;
15         count += 1;
16
17         if (number != SENTINEL) {
18             std::cin >> number;
19         } else {
20             no_sentinel = false;
21         }
22     }
23
24     int average = total / count;
25     std::cout << "Average: " << average << std::endl;
26
27     return 0;
28 }
```

2.2 Python Tasks

1. Consider the below program. What does the program do?

```
1 a = int(input())
2 b = int(input())
3
4 c = a
5 a = b
6 b = c
```

2. Consider the below program. What does the program do?

```
x = 0

number = int(input())
while number != -1:
    if x < number:
        x = number

    number = int(input())

print(x)
```

3. Implement the below algorithm as a program.

```
read in a number N

initialize number P
set P equal to 1

initialize number J
set J equal to 1

repeat the following until J is greater than N:
    set P equal to P multiplied by J
    increment J by 1

print P
```

4. Design and implement a program reading integers from standard input until the value 99,999 is encountered. Print the average of the integers encountered before 99,999 to standard output. Ensure your program works for all inputs.

5. Aubrey, a CSCI 1170 student, was asked to design and implement a program according to the following prompt.

   > Design and implement a program reading integers from standard input until the value 99,999 is encountered. Print the average of the integers encountered before 99,999 to standard output. Ensure your program works for all inputs.

   Below is the program Aubrey designed and implemented. Why is Aubrey's program correct/incorrect?

```
1  SENTINEL = 99999
2
3  total = 0
4  count = 0
5
6  number = 0
7  no_sentinel = True
8
9  number = int(input())
10 while no_sentinel:
11     total += number
12     count += 1
13
14     if number != SENTINEL:
15         number = int(input())
16     else:
17         no_sentinel = False
18
19 average = total // count
20 print("Average:", average)
```

Office of Research Compliance
2269 Middle Tennessee Blvd.
Sam H. Ingram Bldg (ING) Room 010A
Box 124
Murfreesboro, TN 37132
www.mtsu.edu/irb

Date: May 28, 2024
PI: Carlos Aldana Lira
Department: Middle Tennessee State University, Chemistry
Re: Initial - IRB-FY2024-203
Novice Preconceptions in an Introductory Programming Course

The Middle Tennessee State University Institutional Review Board has reviewed and approved by Expedited Review the above referenced research study. The approval is effective starting May 28, 2024.

Decision: Approved
Category: 6. Collection of data from voice, video, digital, or image recordings made for research purposes.
7. Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies. (NOTE: Some research in this category may be exempt from the HHS regulations for the protection of human subjects. 45 CFR 46.101(b)(2) and (b)(3). This listing refers only to research that is not exempt.)

Findings:
Research Notes:

The following apply to your approved study:

1. In accordance with 45 CFR 46.110 and the regulations for Expedited Review (Common Rule), this project does not expire and continuing review is not required by the IRB.
2. Any unanticipated harm to participants or adverse events must be reported to the Office of Compliance.
3. All modifications to the approved study must be submitted for review through Cayuse IRB for approval before their implementation. Adding new researchers constitutes a modification to the protocol. Per MTSU Policy, a researcher is defined as anyone who handles the data or interacts with participants. Everyone meeting this definition for this project must have completed the required CITI training and received IRB approval prior to becoming actively involved in the project.
4. Closure of the study must be submitted within Cayuse when the study ends or when personal identifiers are removed from the data and all codes and keys are destroyed.
5. All research materials must be retained by the PI for at least three (3) years after study completion and then destroyed in a manner that maintains confidentiality and anonymity.

6. All approval letters and study documents are located within Submission Details in Cayuse IRB.

Sincerely,

The Middle Tennessee State University Institutional Review Board