Applications of Out-of-Domain Knowledge in Students' Reasoning about Computer Program State

By

Colleen Marie Lewis

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Science and Mathematics Education

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Andrea A. diSessa, Chair
Michael Clancy
Kathleen Metz

Fall 2012

www.manaraa.com

UMI Number: 3555787

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 3555787

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

www.manaraa.com

www.manaraa.com

Abstract

Applications of Out-of-Domain Knowledge
in Students' Reasoning about Computer Program State

by

Colleen Marie Lewis

Doctor of Philosophy in Science and Mathematics Education

University of California, Berkeley

Professor Andrea A. diSessa, Chair

To meet a growing demand and a projected deficit in the supply of computer professionals (NCWIT, 2009), it is of vital importance to expand students' access to computer science. However, many researchers in the computer science education community unproductively assume that some students lack an innate ability for computer science and therefore cannot be successful learning to program. In contrast, I hypothesize that the degree to which computer science students make productive use of their out-of-domain knowledge can better explain the range of success of novices learning to program. To investigate what non-programming knowledge supports students' success, I conducted and videotaped approximately 40 hours of clinical interviews with 30 undergraduate students enrolled in introductory programming courses. During each interview, a participant talked as they solved programming problems, many of which were multiple-choice problems that were highly correlated with success on an Advanced Placement Computer Science exam. In the analysis of the interviews I focused on students' strengths rather than the typical decision to focus on students' weaknesses. I documented specific competencies of the participants and applied analytic tools from the Knowledge in Pieces theoretical framework (diSessa, 1993) to attempt to understand the source and nature of these competencies. I found that participants appeared to build upon several kinds of out-of-domain knowledge. For example, many students used algebraic substitution techniques when tracing the state of recursive functions. Students appeared to use metaphors and their intuitive knowledge of both iteration and physics to understand infinite loops and base cases. On the level of an individual students' reasoning, a case study analysis illustrated the ways in which a participant integrated her linguistic knowledge of "and" into her reasoning about the computer science command "and." In addition to identifying these specific applications of out-of-domain knowledge, this dissertation applies learning theories that had not previously been applied to computer science education. Through this application I extend the learning theories to the domain of computer science, propose refinements to the theories, and provide insights into participants' reasoning about particular computer science topics.

www.manaraa.com

www.manaraa.com

Acknowledgements

I would like to thank my advisor Andy diSessa whose patience and encouragement has helped me develop my ideas and passion for education research. I came to graduate school wanting to be the best possible computer science educator and Andy has inspired me to aim much higher, to change how people conceptualize computer science learning. He taught me by example to give my research participants the type of respect and thoughtful consideration of their ideas that he gives his advisees and research participants alike. He taught me the importance of every word I use to express my ideas and my writing and thinking are changed for the better.

I would like to thank my advisor Mike Clancy for his endless encouragement. We share a passion for understanding our students and connecting them with experiences that will develop their knowledge and interest in computer science. He matched my enthusiasm for my research questions of the week, was my tour guide of computer science education research, and was a continual source of support and encouragement. Mike expected me to accomplish things that I thought graduate students could not do and I credit his high expectations as the catalyst for my professional accomplishments.

I would like to thank Kathleen Metz for providing the foundation for my engagement in qualitative research. She continually challenged me to align my research goals, analytic method, and the concerns of the educational research community. She pushed my thinking forward by challenging my assumptions and suggesting techniques to improve the validity of my findings.

I would like to thank all of my family and friends who have provided endless support, both emotional and intellectual. In particular I would like to thank Paul Bruno, Katherine Lewis, Cynthia Sturton, Nathaniel Titterton, and Dan Garcia who have tirelessly encouraged, critiqued, and supported me.

The goal of my research, teaching, and volunteer work is to make computer science accessible. I greatly appreciate the people in my life who worked to make computer science accessible to me. In addition to the many excellent instructors who educated and inspired me, including Dan Garcia, Katherine Yelick, and Eric Brewer, I would like to thank Irene Jung for seeing my passion for computer science and reminding me of it anytime I felt discouraged.

www.manaraa.com

Table of Contents

INTRODUCTION 1
Obstacles and Opportunities 1
Dissertation Overview 6
THEORETICAL FRAMEWORK 8
Theory Development 8
Epistemological Commitments 9
Coordination Class Theory and Theoretical Constructs 11
Conclusion 16
METHODS 17
Participant Recruitment 17
Participants 17
Data Collection 18
Sample Size 18
Interview Protocol 19
Analysis Methods 20
Recursion Background 21
Interview Questions 22
THE COORDINATION CLASS OF STATE 34
Methods 39
Case Study: Megan 42
Discussion 55
PARTIAL DESCRIPTIONS OF STATE CHANGE 59
Case Study 61
Previous Research 64
Types of Partial Descriptions of State Change 66
Analysis 67
Conclusions 72
INTUITIVE KNOWLEDGE ABOUT BASE CASES AND INFINITE LOOPS 74
Motivation: No Evidence of the use of a Memorized Response 77
Previous Research 79
Hypotheses Regarding Infinite Loop Knowledge 81

www.manaraa.com

Hypotheses Regarding Base Case Knowledge 83
Conclusion..85
SUBSTITUTION TECHNIQUES 87
Methods..90
Substitution Technique: Simulating Execution 91
Substitution Technique: Accumulating Pending Calculations 93
Substitution Technique: Memoization 95
Substitution Technique: Solving it by hand 96
Discussion 99
CONCLUSION 103
The coordination class of state 104
Partial Descriptions of State Change 104
Intuitive knowledge about base cases and infinite loops 105
Substitution techniques...106
Summary of Contributions 106
REFERENCES...107

www.manaraa.com

INTRODUCTION

It is common for a college course to be a student's introduction to programming. Many students come with enthusiasm, motivation and a track record of academic success. However, despite the best efforts of the student and the instructor, many students appear to never "get it" (McCracken *et al.*, 2001).

It is an open question what non-programming experiences may support success learning to program (Simon *et al.*, 2006). In this dissertation I investigate the question of what experiences students bring to the computer science classroom, how they can contribute to success, and how computer science pedagogy can take advantage of them. There is strong support for the assumption that without understanding the interplay between non-programming knowledge and the learning of programming, pre-programming and programming instruction at best will be impoverished and at worst will fail (Soloway, Bonar, & Ehrlich, 1983; Fleury, 1993; Ben-Ari, 2001; diSessa, 1986; Vosniadou & Brewer, 1992; diSessa, 1993; diSessa & Wagner, 2005).

In this project I investigate the constructivist assumption that prior knowledge can serve as a significant support for learning computer programming. I hypothesize that the degree to which computer science students make productive use of their out-of-domain knowledge can explain the range of success of novices learning to program. A common (Robins, 2010) alternative assumption within the computer science community is that innate aptitude for computer programming explains the range of students' success: people are born as programmers or non-programmers (Dehnadi, 2006; Lister *et al.*, 2004; Reges, 2008; Simon *et al.*, 2006). According to the work of Dweck (2007) and Steele (1997), when this assumption underlies pedagogy, student learning and attitudes suffer.

To investigate what non-programming resources and non-programming strategies support students' success, I conducted a detailed analysis of student reasoning on computer programming questions that were identified by previous research (Reges, 2006) and will be discussed at greater length in this chapter and the methods chapter.

The goal of this line of work is to transform computer science education through identifying and building upon students' strengths to ultimately support the success of more students. This is of vital importance to increase access to computer science and to meet a growing demand and a projected deficit in the supply of computer professionals (NCWIT, 2009).

Obstacles and Opportunities

Obstacles

I hypothesize that there are two related obstacles to the success of introductory programming courses. The first is the belief of many computer science students that computer science is unrelated to their previous experience and ways of thinking. The second is the belief of many computer science instructors and many computer science education researchers that

www.manaraa.com

success in computer science is determined by innate ability. Both are counterproductive for student learning as is elaborated below (Dweck, 2007; Steele, 1997).

Student Beliefs

In a related project (Lewis, Yasuhara, & Anderson, 2011), we found that students frequently describe computer science as unconnected to their previous ways of thinking. For example, one student said, *"It's like a different way of thinking. Like it's really confusing. You have to get used to it."* Another similar sentiment, *"I feel like you shouldn't do it unless you like—unless you're like more attuned to that kind of thinking. If you don't think that way, it's just going to be really difficult for you."* Students appear to believe that their existing ways of thinking are not relevant to learning to program and that to be successful in computer science they have to adopt a completely new way of thinking. This model of adopting a new way of thinking rather than adapting your current thinking may be a significant barrier to students making productive use of prior knowledge.

Educational research from a wide variety of fields argues that students' prior knowledge must be taken into account (Ben-Ari, 2004; diSessa, 1993; Fleury, 1993; Soloway, Bonar, & Erlich, 1983; Vosniadou & Brewer, 1992). Pennington (1987) found that the most successful programmers were those who frequently made connections between the program text and the non-programming or real-world goals. Based upon this finding, I hypothesize that students will be less successful if they fail to connect their programming knowledge to their prior non-programming knowledge.

Instructor and Researcher Beliefs

The hypothesis that students have untapped resources upon which we can transform undergraduate computer science education runs counter to what may be a common assumption among computer scientists of the existence of an innate aptitude that determines students' success learning to program (Robins, 2010; Lewis, 2007). While it is possible that students could have a genetic predisposition to program computers, this is currently an untested assumption, which can have real consequences and can play into self-fulfilling prophecies (Dweck & Legget, 1988; Steele, 1997). Even if we assume that many students lack the intellectual resources to become as successful as Alan Turing, we hope to connect all motivated students with an environment in which they can become competent at programming. In contrast, Simon *et al.* (2006) summarize an ongoing research direction that we believe may be an outgrowth of a dearth of explanations of why some students are less successful.

> "The literature abounds in assertions of the existence of an aptitude for programming, and of attempts to find a suitable predictor for that aptitude so as to avoid wasting time and effort educating students who are unlikely ever to become good programmers." (Simon *et al.* 2006)

The assumption of an innate aptitude is often implicit and is made explicit in more subtle ways. For example Lister *et al.* (2004) describe why differences in innate talent at various institutions constitute a complication in analyzing the study's multi-institutional data.

www.manaraa.com

> “Clearly, some institutions attract students with a greater innate talent for programming.” (Lister *et al.*, 2004)

In multi-institutional studies it is arguably relevant to discuss differences in the student populations. However in this statement the authors indicate an otherwise unstated assumption that there exists an “innate talent for programming”. Barker *et al*. (Barker, McDowell & Kalahar, 2009) demonstrates a more subtle instantiation of this assumption of innate aptitude in appealing to the idea of “weeding out” students.

> “Introductory classes should weed students out based on ability and potential, not on the weight of the workload” (Barker, McDowell & Kalahar, 2009).

In this and her other work (Garvin-Doxas & Barker, 2004), Barker attempts to direct computer science educators to practices that will support an inclusive and non-defensive climate within the computer science classroom. The juxtaposition of the goals of Barker’s research and the seeming acceptance and endorsement of “weed[ing] students out” suggests the prevalence of the belief in an innate aptitude for programming.

In another example, Dehnadi (2006) bemoans the fact that students are not afforded the opportunity before college to be “streamed” into those that “can” and “can’t” be successful.

> “Part of the problem is that the subject is not widely taught at school, so undergraduates arrive without having being streamed into those who can do well and those who can't.” (Dehnadi, 2006 p. 53)

An important aspect of the language here is Dehnadi’s use of the words “can” and “can’t.” As a practical point we have computer science students that “are” and “are not” successful. However this subtle difference between “are not” and “cannot” appears to represents a core assumption of innate aptitude.

Clayton Lewis (2007) investigated the prevalence of various beliefs amongst computer science professors and students. He found that 10 out of 13 professors surveyed rejected the statement “Nearly everyone is capable of succeeding in the computer science curriculum if they work at it.”

In my related research, I have documented students’ beliefs about whether or not computer science ability is innate and how the environment of an introductory programming class shapes these beliefs (Lewis, Yasuhara, Anderson, 2011). There was variation between participants’ beliefs; some participants rejected and some students endorsed the existence of an innate ability for computer science. There were cases in which participants endorsed the existence of an innate ability that demonstrated how this belief can discourage persistence and exclude students that are underrepresented within computer science. The student quoted below attributed the idea that computer science ability is innate to her introductory computer science professor. I interpret her statements as suggesting that students’ difficulty in computer science can be attributed to an unchangeable lack of innate ability. She said:

www.manaraa.com

> *"Even my [UA-CS2 professor] told us that some people are just born that way, with that mental outlook that is compatible with CS… They feel it's so easy for them... Yeah, and he told the rest of the people that some of you will try but some of you won't get it, and it's just that your mental outlook isn't made that way. It's something you're born with. You can't help it"* (p. 6, Lewis, Yasuhara, & Anderson, 2011)

Another Participant in this study said that she thought female students might be less innately abled at computer science and said that few women in the field might be evidence of this lack of an innate ability (Lewis, Yasuhara, & Anderson, 2011).

Based upon research from Dweck and her colleagues (see Dweck & Leggett, 1988 for a review) and Steele and his colleagues (Steele, 1997; Carr & Steele, 2009) there are negative consequences for students when their success or lack of success is framed as indicative of innate aptitude.

Carol Dweck and colleagues (*e.g.*, Dweck & Leggett, 1988) have researched how students behave when reasoning with a fixed or growth mindset. A fixed mindset views intelligence as static while a growth mindset views intelligence as malleable. For example, when students read a passage where intelligence was defined as innate, the students were less likely to choose challenging tasks than students who were presented with a text that defines intelligence as malleable (Dweck & Leggett, 1988). If students come to believe that there exists an innate aptitude for computer science they may adopt a fixed mindset, which can stifle their academic growth (Dweck & Leggett, 1988; Simon *et al.*, 2008).

Claude Steele and his colleagues (Steele, 1997; Carr & Steele, 2009) have identified a related phenomenon named stereotype threat. Consider the stereotype of women being bad at math. Spencer, Quinn and Steele (1999) gave two groups of men and women a math test. The first group was told that the test was diagnostic of ability and that women tended to perform poorly on the test. The second group was told that the test was not diagnostic of ability and that men and women tended to perform equally well. In this and other studies (e.g. Steele & Aronson, 1995), the stereotyped group performed less well than their non-stereotyped peers only in the group that was told that the test was diagnostic of ability.

Steele (1997) explains that when members of a stereotyped group come to believe that a stereotype could be used to interpret their performance, their behavior tends to reinforce the stereotype. It is not necessary that an individual believes the stereotype to be true, only that the stereotype is activated and reflects upon a domain with which he or she is identified (Steele, 1997). This phenomenon has held for stereotypes of the intellect of black students (Steele & Aronson, 1995), stereotypes of the intellectual inferiority of white males to Asian males (Aronson *et al.*, 1999) and stereotypes of white people as racist (Goff, Steele, & Davies, 2008). Therefore if students who are invested in their success in computer science believe that stereotypes of their abilities in computer science are relevant, their performance may be artificially depressed.

www.manaraa.com

Opportunities

The research in this dissertation sits at a crossroad of opportunity. Below I discuss how the proposed research takes advantage of the frequently late introduction of computer science, builds upon successful research and pedagogy efforts in physics education, and capitalizes on previous research that has identified central multiple-choice computer science problems (Reges, 2008) in the highly interconnected domain of computer science (Robins, 2010).

Late Introduction to Computer Science

Few students have the opportunity to learn computer science before attending college. Unlike other intellectual domains, such as mathematics or history, many students' first introduction to computer science is in college. Certainly some students have access before college, but the inequality of access then creates a heterogeneous population of students with prior experience at the college level. For example, in 2010, only 19.2 percent of the Advanced Placement Computer Science (AP CS) test takers were female (The College Board, 2011). In 2010, this was the lowest ratio of female-to-male test-taking rates of any of the offered Advanced Placement tests. AP CS courses are not the only computer science courses offered at the pre-college level, but the data regarding test-taking patterns for the AP CS exam suggest that female students will be overrepresented in the population of students that do not have programming experience before college, which has been observed at the University of California, Berkeley (Lewis, Titterton, & Clancy, 2012).

As documented by Margolis and others (Margolis *et al.*, 2008) few students have experience learning computer science before college. However, with this missed opportunity of early learning comes a unique opportunity for educational research. As an educator and an educational researcher I often work with students who are first learning computer science in college and therefore I have the opportunity to observe students engaged in learning within a domain for which they are both ignorant and potentially well prepared by their other academic experiences. However, some students with what we believe to be adequate preparation and motivation are not successful. We do not know what academic experiences would enable someone to be well prepared for the learning of computer science.

Evidence of Success in Other Domains

The proposed research follows a long line of research investigating prior knowledge from other domains (diSessa, 1993; diSessa & Sherin, 1998; Wittman, 2001; diSessa & Wagner, 2005; Wagner, 2006; Parnafes, 2007; Levrini & diSessa 2008; Hammer, 2000; Russ & Sherin, 2008) and developing transformative pedagogy (diSessa & Minstrell, 1998). This prior work provides methodological examples of how to identify students' knowledge resources and beliefs that play a role in learning. More generally these researchers engage in the enterprise of studying conceptual change and attempt to understand the dynamic process of thinking and learning in the domain of physics. The current study builds upon this work to consider conceptual change within computer science education.

www.manaraa.com

Questions Central to Computer Science Competence

Robins (2010) presents a similar critique of the computer science community's assumptions of the existence of an innate aptitude for programming. He claims that this assumption is fueled by instructors' experience of a bimodal distribution of student grades in introductory programming courses. While Dehnadi (2006) claims that this bimodal result is because the course separates "those who can do well and those who can't" (Dehnadi, 2006 p. 53), Robins (2010) claims that it is the highly connected nature of the domain of computer science that produces the bimodal distribution. Robins (2010) builds a constructivist model of learning, such that a student's failure to grasp a concept early in the course negatively influences his or her chance of understanding later concepts. Building this alternate assumption into a computational model, Robins (2010) was able to simulate the bimodal grade distribution patterns observed by instructors. From Robins' (2010) work, I take the hypothesis that computer science may be a highly interconnected domain. From this hypothesis, I attempt to identify questions that capture central connections in the domain and turn to the work of Reges (2008) to identify some such questions.

Reges (2008) analyzed results from the 1988 AP CS exam. He found that five multiple choice questions accounted for the majority of the pair-wise correlations between multiple choice and free-response questions on the exam. Reges (2008) frames the question exposed by his research in the following quote.

> "do [the highly correlated questions] measure a fundamental ability that some people have more than others? If so, can that ability be effectively tested before a student takes a course?"

Using clinical interviews I captured novice programmers' answers to these highly-correlated questions from the 1988 AP CS exam. However, this dissertation does not seek to answer the question of whether "a fundamental ability that some people have more than others" can be "tested before a student takes a course." Instead I see to answer the question of whether as educators we can help students develop that ability. Instead of investigating a static "fundamental ability" I investigate whether these questions may measure teachable competencies that may be identified and explored by analyzing how students reason about these questions.

Dissertation Overview

In response to the lack of factors that can predict success learning to program, I hypothesize that students' success is shaped not simply by having a particular non-programming competence, such as a skill or set of skills from math, but the degree to which students make productive use of their non-programming competence when learning to program. The goal of this dissertation is to investigate the hypothesis that students have out-of-domain knowledge that is relevant to the learning and doing of computer programming and to develop hypotheses about the content and function of that out-of-domain knowledge.

An emphasis throughout the dissertation is novice programmers' understanding of computer program state because it is a language independent description of some of the key

www.manaraa.com

competencies of programming. Computer program state is the set of all information calculated and maintained by the machine when executing a program. This includes user-defined variables, arguments to functions, return values from expression and sub-expressions, and stack information such as the program counter and nesting of function class. Numerous researchers have emphasized the importance of program state (du Boulay, O'Shea, & Monk, 1989; du Boulay, 1989; Shinners-Kennedy, 2008; Papert 1980; diSessa, 2000; Cooper, Dann, & Pausch, 2000).

The analysis chapters each provide an additional perspective on novice programmers' reasoning about computer program state. These analyses were inspired by observations of competence amongst the research participants. Upon documenting these competencies, I evaluated various analytic tools for exploring the nature and source of the competence.

This data collection and much of the analysis is informed by the Knowledge in Pieces theoretical framework. The following Theoretical Framework chapter provides an overview of relevant details from the Knowledge in Pieces theoretical framework. This includes two models of knowledge that I apply within my analysis and based upon analysis I extend and refine. The first theory is a model of a particular type of conceptual knowledge referred to as coordination class theory (diSessa & Sherin, 1998) and the second theory is a model of a type of intuitive knowledge referred to as p-prim theory (diSessa, 1993).

www.manaraa.com

THEORETICAL FRAMEWORK

This chapter provides details regarding the Knowledge in Pieces theoretical framework, which has shaped the data collection and analysis in this study. I begin with background regarding the enterprise of theory development particular to the Knowledge in Pieces theoretical framework. Next I present details of Knowledge in Pieces including the foundational epistemological commitments and a theoretical model of a particular type of concept, referred to as a coordination class. All of the analysis in this dissertation is built upon these epistemological commitments and the first analysis chapter applies coordination class theory to computer science for the first time. This section is intended to provide relevant background regarding Knowledge in Pieces, coordination class theory, and the style of research and theory development undertaken in this dissertation.

Theory Development

While someone might colloquially say that an individual has a concept, this everyday notion of a "concept" specifies very little about the individual's knowledge. Researchers contributing to the Knowledge in Pieces theoretical framework develop models of learning and knowing that go beyond typical dictionary definitions. To introduce the content of these models I separate the features of knowledge that are described as either about the observable behavior, content and form, or dynamics of a person's knowledge. This is not a traditional segmentation of the research, but attempts to highlight the scope and focus of Knowledge in Pieces research.

I define the observable behavior of knowledge as the observable aspects of an individual's knowledge. This includes coarse measures such as whether or not an individual answers a question correctly. This also includes subtle features in the content of an individual's explanation or answer to a question, such as the details of their solution path. The behavior of an individual's knowledge is a typical focus of educational research. This is an important aspect of knowledge to emphasize, but this study and others within the Knowledge in Pieces line of work shift the focus to the content, form, and dynamics of an individual's knowledge.

Work from within the Knowledge in Pieces line of work emphasizes these two other aspects of knowledge: the content and form of knowledge and the dynamics of knowledge. Specifying the content and form of knowledge is primarily a theoretical task in which the researchers attempt to build a model of the types and properties of an individual's knowledge. Developing a model of the content and form of knowledge goes hand in hand with developing a model of the dynamics of knowledge. I define the content and form of knowledge as the content of specific knowledge and hypothesized forms of this knowledge while the dynamics of knowledge specifies how various knowledge resources interact to produce the observable behavior of knowledge.

Specificity in the definitions of these terms and predictions of the model is necessary to provide for the possibility of rejecting or identifying necessary changes in the theory presented. To discuss and develop theories that specify the content, form, and dynamics of knowledge it is

www.manaraa.com

not sufficient to use everyday labels of knowledge such as “concept” or “understanding.” These everyday labels of knowledge are too coarse to describe the content, form, and dynamics of knowledge that could produce the diversity of behavior observed. A major emphasis in coordination class theory is to move beyond typical definitions of terms for describing learning. In particular, coordination class theory is specific about what it means to have a particular concept.

The current study, and much of the work that has adopted the Knowledge in Pieces theoretical framework, attempts to develop theories of learning. The Knowledge in Pieces theoretical framework is comprised of a number of overlapping theories of learning. These theories are informed by observing individuals’ reasoning about various situations. From these data, models regarding the content, form, and dynamics of knowledge are developed to match the observed data. These models are the primary component of the theories of learning and are taken as works in progress that are continually refined (Cobb, Confrey, diSessa, Lehrer, & Schauble, 2003). For example, a model of a particular type of concept, referred to as a coordination class, has been continually expanded and refined by subsequent studies (diSessa & Sherin, 1998; Wittman, 2001; diSessa & Wagner, 2005; Wagner, 2006; Parnafes, 2007; Levrini & diSessa 2008) These continually refined models, which are often referred to as theories, are different from the commonly referenced theories in physics. In physics, theories are typically static and infrequently questioned. The theories in the Knowledge in Pieces line of research can be seen as earlier in the process of theory development. In this line of work, theories are intended to be developed, scrutinized, extended, and refined. This iterative process of development both changes and improves these theories (Cobb, Confrey, diSessa, Lehrer, & Schauble, 2003).

Epistemological Commitments

While the theories that comprise the Knowledge in Pieces theoretical framework are actively refined, there exist some conclusions regarding the content, form, and dynamics of knowledge that are shared by researchers who apply this theoretical framework. I preface my introduction of coordination class theory by identifying some of these common epistemological commitments in research using the Knowledge in Pieces theoretical framework.

I will refer to the components of an individual’s knowledge as knowledge resources. While the language varies between researchers (diSessa, 1993; Hammer, Elby, Scherr, & Redish, 2004), these knowledge resources are not assumed to be encoded in a uniform way (diSessa, 1993). As an illustration of the diversity of encodings, I will describe two possible encodings of knowledge that govern the opening of jars. As a first example, I can easily recall and interpret the phrase “righty-tighty, lefty-loosey.” Individuals may have factual knowledge like this encoded as a particular phrase. This can be seen as a different type of knowledge than the knowledge I use when I, without recalling the phrase, reach to untwist a jar lid. We can think of these two knowledge resources as being of a similar grain size because they both govern the opening of jars, but they are almost certainly encoded in different ways. It is likely that the first is primarily encoded as a phrase while the second is primarily encoded as what would colloquially be referred to as muscle memory.

www.manaraa.com

In addition to assuming a diversity of encodings of knowledge resources, the Knowledge in Pieces theoretical framework specifies that various knowledge resources can work together to produce more complex competence (diSessa, 1993). This implies that reasoning patterns that can be observed in behavior are sometimes supported by not just a single knowledge resource, but a network of knowledge resources. "Knowledge in Pieces" refers to this network of knowledge resources that are assumed to support everyday and scientific reasoning. For example, compare the knowledge resources to open a jar with the knowledge resources involved in a more complex task such as replacing the brakes on a bicycle. We can see this more complex task as requiring more knowledge than is necessary to open a jar. Replacing the brakes on a bicycle may even require some subset of the knowledge needed to open a jar. While we could model this as a single knowledge resource with larger scope, it might better be described as, itself, a collection of knowledge resources.

For a non-expert, the application of these knowledge resources is frequently described as an emergent process and that the dynamics of knowledge and the details of the situation influence the particular application of knowledge. This can result in the observable fact that the knowledge an individual applies in a context may vary and can explain a lack of coherence that has been observed in various studies of novice knowledge (Kahney, 1989; Vosniadou & Brewer, 1992). For example, Kahney (1989) found that some students were inconsistent in their predictions of the behavior of recursive function calls. While many researchers presume that students have definite models of particular concepts (see diSessa, 2006), this does not explain some of the behavior of students' knowledge such as answering correctly on one question while seeming unable to produce the same performance on another question. This can be explained by the presence of a diversity of knowledge resources, which are not uniformly applied to produce expert performance.

In the Knowledge in Pieces theoretical framework (diSessa, 1993), whether or not a piece of knowledge is accessed by an individual in a context is referred to as whether or not that piece of knowledge is *cued* or *activated*. This construct of cueing was introduced to describe the relative priority of a type of intuitive knowledge within an individual's knowledge system (diSessa, 1993), but has not been used in coordination class theory research. A description of this type of intuitive knowledge will be provided later in this chapter. I reference it here only to note that I will apply this language of *cued* and *activated* to discuss a greater diversity of knowledge resources because it is consistent with the epistemological commitments of coordination class theory.

Each knowledge piece can be thought to have a particular priority of cueing for each context, referred to as its *cueing priority*. Knowledge can be cued by elements in the external environment or be part of a network of closely connected knowledge elements that are cued together. Cueing priority might be viewed as a measure of an individual's unconscious assumption regarding the applicability of that knowledge in a context. This explains some examples of failure of transfer, where an individual has demonstrated use of some knowledge that they appear to not apply in a new context (diSessa & Wagner, 2006). Knowledge that is cued is available to the individual, but the individual may decide that the knowledge is not

www.manaraa.com

relevant to a particular context. This phenomenon is analyzed in previous research (diSessa & Sherin, 1998; Wagner, 2006) and my first analysis chapter.

A specific instantiation of this idea of diversity of encodings and network of knowledge can be found in diSessa's model of intuitive knowledge (1993). This exemplifies a final epistemological commitment that individuals' everyday knowledge interacts with academic knowledge in individuals' reasoning. diSessa (1993) identified a class of knowledge resources that he referred to as phenomenological primitives, or p-prims. P-prims are hypothesized to be a primitive knowledge resource in an individual's knowledge system, meaning that p-prims are not composed of more primitive knowledge resources. P-prims are phenomenological, meaning that they relate to physical phenomena in the world. P-prims are presumed to be responsible for some of individuals' expectations regarding physical phenomena. For example, diSessa (1993) identified and labeled Ohm's p-prim as the knowledge resource responsible for the intuition that you have to work harder to push a heavy shopping cart than to push a light shopping cart. diSessa (1993) schematized this intuition from Ohm's p-prim as that "[a]n agent or causal impetus acts through a resistance or interference to produce a result" (p. 217, diSessa, 1993). This intuition relates to Ohm's law: instead of relating voltage, current, and resistance, Ohm's p-prim relates effort, result, and resistance.

In summary, the Knowledge in Pieces theoretical framework assumes that students have a diverse set of knowledge resources available to them, which includes knowledge from in- and out-of-school settings. These knowledge resources interact to produce individuals' observed behavior and these knowledge resources are assumed to be varied in encoding. Depending upon details of the situation an individual may invoke different resources.

Coordination Class Theory and Theoretical Constructs

The Function of Coordination Classes

In everyday use there is a diversity of things that count as concepts. For example, we could label "surface area," "chair," and "love" all as "concepts." These are each different ideas with different ways of knowing. Certainly the way in which individuals could demonstrate that they have the concept of "surface area," "chair," and "love" differs between these three concepts. For example, an expert with the concept of surface area might be able to identify the surface area of various objects while an expert with the concept of chair may be able to identify whether various items are in fact chairs. As a rough approximation, expertise with surface area involves measurement and calculation while expertise with chairs involves classification. Measurement and calculation are dissimilar in many ways to classification. I will not speculate how an expert with the concept of love might demonstrate that expertise, but it is likely different than the competence associated with "surface area" or "chair." This motivates why moving beyond colloquial terms for learning such as "concept" may provide clearer and more coherent constructs for theories of learning.

Given the difference in how these competencies are demonstrated, it may be inaccurate to assume that the development of these concepts is uniform or that the dynamics of use of these concepts is uniform. diSessa and Sherin (1998) specified that a coordination class is a

www.manaraa.com

model of only a particular set of concepts in an attempt to model concepts that have more uniform development and dynamics of use. Therefore, coordination classes do not include everything that would colloquially be referred to as a concept, but instead a set of concepts that are expected to be more similar in development and use. Including only a subset of all possible concepts allows for greater coherence between examples of coordination classes. The operational definition of a coordination class requires that the concept have a particular functional role in the individual's reasoning, which I elaborate below.

Before describing the general qualifications for what concepts are classified as a coordination class, I will describe a few examples of coordination classes. The canonical coordination class discussed by diSessa and Sherin (1998) is force. They specify that the primary function of this coordination class is to precisely identify forces in the world. This includes identifying components of those forces such as position, direction, and magnitude. In this case, each of these components is also a separate coordination class. For the coordination class of position, the primary function is to precisely identify positions in the world. Identification of positions could take place in the physical space or within a representation of space such as a graph. These examples are intended to show that applying a coordination class can involve identifying a complex set of information, such as the components of force, and possibly doing so across contexts, such as physical space and graphs.

A coordination class is a model of knowledge in which the primary function of the coordination class is to identify a type of information in the world, like force in the previous example. Throughout the description of coordination classes, I will draw on examples from the coordination class of surface area. It follows from the definition of coordination classes that the primary function of the coordination class of surface area is to identify surface areas in the world. Surface area will be discussed here as a simple example.

Coordination classes, unlike p-prims, are not assumed to be primitive knowledge resources. In fact, coordination classes refer to the collection of knowledge resources that work together to produce a particular competence. The model of coordination classes specifies that these component knowledge resources form connections that govern what knowledge is used together.

Coordination and Problems with Span and Alignment

Coordination class theory characterizes learning as a process of *coordinating* what knowledge should and should not be applied within a context (diSessa & Sherin, 1998). Coordination class theory specifies two main challenges in this process of coordinating knowledge.

The first main challenge is an issue of *alignment*, which is essentially a measure of whether an individual can correctly determine the relevant information of the coordination class (diSessa & Wagner, 2005). This requires that for a given context, the individual correctly determines the focal information of the coordination class. Sometimes there may be multiple ways of determining the focal information. Alignment describes cases in which the individual correctly determines the focal information regardless of the method and knowledge employed.

www.manaraa.com

For example, there may be two ways to determine surface area, each using different knowledge resources. An individual has adequate alignment in a situation if when using either method he or she arrives at the same correct answer (diSessa & Wagner, 2005).

The second main challenge relates to the difficulty of recognizing the relevance of knowledge across contexts, which is referred to as problems with *span*. Consider the following three examples of student difficulties in applying a coordination class. The first two are examples of a problem of span, while the third is an example of a problem with alignment.

- The individual incorrectly believes that his or her relevant knowledge is not applicable in the given situation (lack of span).
- The individual recognizes the relevance of his or her knowledge, but does not know how to identify the information in the given situation (lack of span).
- The individual believes that he or she has relevant knowledge, but he or she identifies the information incorrectly in the given situation (lack of alignment).

Research using the Knowledge in Pieces theoretical framework typically focuses on individuals that could be considered novices in the topic domain. However, coordination class is the label for the expert form of knowledge, which appropriately uses knowledge resources to produce the desired competence. Therefore, in a true coordination class, this is to say, in the knowledge system of an expert with that particular concept, different types of knowledge work together to accurately identify the focal information across all applicable contexts. A coordination class is an ideal. We cannot demonstrate that an individual possesses this knowledge, we can only identify individuals that do not possess this coordination class by demonstrating problems of span or alignment.

An individual that has sufficient span and alignment in some context could be described as having appropriate coordination in that context. Appropriate coordination is a determination of the performance of an individual's knowledge in a particular context and does not imply that the individual would have appropriate coordination across all contexts. Thaden-Koch, Dufrense and Mestre (2006) introduced the term "coordination system" as the name for a less than complete coordination class. I will assume that the study's participants have only coordination systems, but I will refer to these as coordination classes as is consistent with much of the previous literature (diSessa & Sherin, 1998; diSessa & Wagner, 2005).

Readout Strategies and Extraction

diSessa and Sherin (1998) define readout strategies as basic perceptual skills for extracting information from the world. For example, when presented with an object, an individual can hold the object and perceive its shape, size, color, texture, or weight. Each of these things that can be perceived is tied to a readout strategy. For example, perceiving color is a readout strategy. diSessa and Sherin (1998) refer to the perception that is generated with one of these readout strategies as a "readout."

www.manaraa.com

Actually, diSessa and Sherin (1998) also use the word “readout” to refer to a more expansive process of making inferences based upon gathering information using these readout strategies. For example, diSessa and Sherin refer to the process of making inferences when they claim that their subject was “reading out the amount of force” (p. 1179) and that the “issue is one of readout” (p. 1180). This double use of the phrase readout was not intended (A. A. diSessa, personal communication, April 3, 2012) and “readout” was intended to mean only the immediate product of using a readout strategy.

For clarity, I will use the term “readout strategy” that has been used consistently, but will not use the term “readout.” I will refer to the result of using a readout strategy as an extraction, which is a term not previously used by coordination class literature. Extractions are made in reference to a particular object. A perception of an object’s color is an example extraction. Readout strategies are strategies that can produce a class of extractions. Readout strategies are general and are not specific to a particular object. Extractions are not general, but an application of a readout strategy in a particular context.

An important aspect of the coordination class model involves the selection of ways of perceiving or extracting information about the world using these readout strategies. In general, we expect individuals not to be limited in executing necessary extractions (diSessa & Sherin, 1998). However, an individual needs to use knowledge to determine what ways of perceiving (or readout strategies) may be relevant.

An individual’s knowledge guides how he or she consciously or unconsciously selects readout strategies that are relevant to the context. For example, when asked to calculate the area of a surface, an individual may pay attention to the height and width of the surface and not the color, because he or she has some knowledge that height and width are relevant to surface area and color is not. However, without this inference he or she might not extract width, which is akin to not attending to that feature in the environment.

Inferences, Causal Net, and Concept Projection

Once an individual has extracted the height and width, he or she must now use other aspects of his or her existing knowledge, such as area = height * width, to make an inference or set of inferences to determine the surface area. Inferences can be built from the information extracted from the world and from existing knowledge. These inferences are defined as taking place in the individual’s *causal net*, which is the term to describe the subset of an individual’s knowledge system that relates to the coordination class (diSessa & Wagner, 2005).

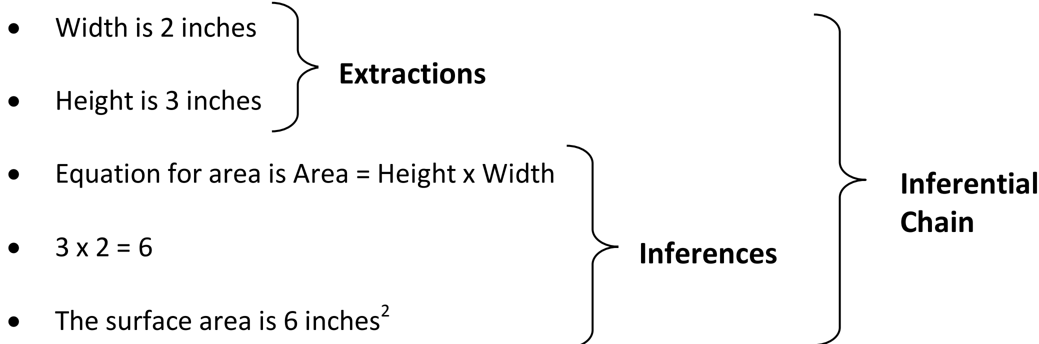
Determining the surface area of a rectangle requires only one relatively simple inference. However, the process of applying readout strategies and generating inferences may be a much more complex, and possibly iterative, process. For example, when an individual calculates the surface area of a shape composed of various triangles and semi-circles, it might not be possible to extract all relevant lengths at once. A single extraction of the radius of one semi-circle may cause an inference that two semi-circles of equal size create a circle. This inference may redirect the individual’s attention to a new extraction, attempting to identify a semi-circle of the same size that could complete the semi-circle. This focused attention to

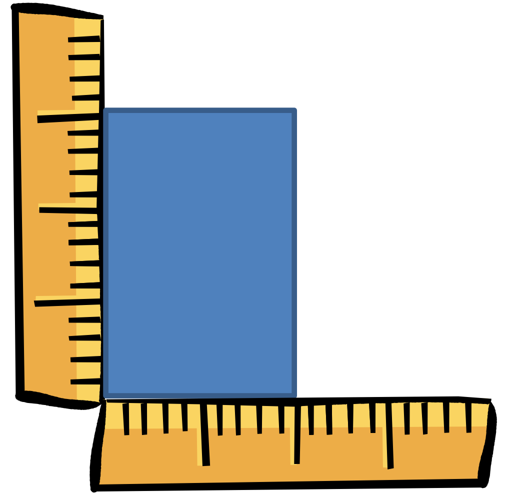
www.manaraa.com

aspects of the environment can be conscious as well as unconscious (Thaden-Koch, Dufrense & Mestre, 2006).

The subset of knowledge from the causal net that an individual uses to identify the focal information of the coordination class in one particular case is referred to as a *concept projection* (diSessa & Wagner, 2005). This includes their chain of inferences, the content of their extractions, and all knowledge that supported the final determination of the focal information. Some of this knowledge is used in a chain of inferences to guide the iterative extractions from the environment. These knowledge resources come from the individual's causal net, which is the sum of the individual's knowledge that is relevant to the coordination class and is not specific to any instance of reasoning.

Figure 1 shows a representation of some of the components of the coordination class model of identifying the surface area of a 3 inch by 2 inch rectangle. As described above, the individual must extract this information from the environment using readout strategies and then develop inferences based upon those extractions and other knowledge. These inferences and the extractions together form an inferential chain, which in **Figure 1** is modeled as being built from the top down. The concept projection is formed by this inferential chain and the readout strategies.





www.manaraa.com

Figure 1. Graphical representation of the coordination class model of one concept projection for surface area.

Conclusion

This section has attempted to motivate the use of non-colloquial terms to describe learning, to exemplify the type of theory refinement I undertake in this dissertation, and to familiarize the reader with coordination class theory and the epistemological commitments of the Knowledge in Pieces theoretical framework.

www.manaraa.com

METHODS

Participant Recruitment

Participants were recruited from the UC Berkeley courses described below via an email recruitment that was forwarded by their course instructor. The body of this email is shown below.

> Hello students in [fill in the course name here],
>
> You are invited to participate in a research project studying how people learn to program. If you chose to participate you will be given a $15 Amazon.com gift card for an hour interview. You may be asked to participate in a second hour long interview. The interviews will take place in Soda hall or Tolman hall and will be scheduled at your convenience. If you are interested in participating, please fill out the form at the following link and a researcher will contact you.
>
> http://www.eecs.berkeley.edu/~colleenl/interview/
>
> Participation in this research will have no bearing on your standing in the class and your instructor will not know which students have chosen to participate.
>
> Thank you,
>
> Colleen Lewis (Graduate Student in Computer Science and Education)

Interested students emailed me their preferred interview times from a list of available interview times listed online. Participants were scheduled on a first-come first-serve basis and all participants who emailed me to schedule an interview were interviewed.

Participants were given a $15 Amazon gift card for participating in the study. In instances where a participant was interviewed multiple times, the participant received one $15 Amazon gift card for each interview.

Participants

Students enrolled in their first programming course were interviewed after they had completed the programming content that is comparable to the content tested on the interview questions. Each course from which students were recruited is described briefly below.

CS10 – Scratch-Based Introductory Programming Course (9 students): The course "The Beauty and Joy of Computing" is the newest addition to UC Berkeley's lower division curriculum (Garcia, Harvey, & Segars, 2012) and uses a modified version of the Scratch programming language (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) that adds functions and lambda (Harvey & Mönig, 2010). The course uses a modified lab-centric

www.manaraa.com

structure (Titterton, Lewis, & Clancy, 2010) with two hours of lecture, one hour of discussion, and four hours of lab a week.

CS3L – Scheme-Based Introductory Programming Course (6 students): UC Berkeley's previous introductory course "Introduction to symbolic programming" using the Scheme programming language (Friedman & Felleisen, 1996) introduces students to basic control structures and recursion. The course uses the lab-centric instruction approach (Titterton, Lewis, & Clancy, 2010), which includes a single hour of lecture a week and six hours of lab.

CS3S – Self-Paced Scheme-Based Introductory Programming Course (15 students): This course is "self-paced" and does not have required class meetings. Students have the option of taking a two- or four-unit version of the course. The four-unit self-paced version covers roughly the same content as CS3L. While interviews took place at the end of the semester, the content covered by individual students varied greatly. This was partially because of the self-paced nature – some students were behind – and partially because some students were only taking 2 units and were required to complete less content throughout the semester. Despite the differences in background, all students that were recruited from this class had seen the relevant content in their course.

Data Collection

Each participant was videotaped solving computer programming problems. The camera was focused on the paper and the area around the paper. The intention was to capture the students' gestures when pointing to text from the problem and their inscriptions on the page. Each student's body and face were not captured so as to provide higher resolution of these gestures and inscriptions. No demographic data was collected from participants.

Sample Size

Six students participated in the pilot round of data collection. All of these students were recruited from a single offering of CS3L. In the first phase of the research, interviews were conducted with seventeen students. Two of these seventeen participants were enrolled in CS10 and fifteen participants were enrolled in CS3S. In the second phase of the research, interviews were conducted with seven students from the introductory programming course using the programming language Scratch.

Students that had performed in the lowest quartile on the first exam in this class were recruited to participate in the second phase of interviews. Four of these students were interviewed more than once. During the first interview in phase 2, participants described how to solve each of the programming problems from the first exam that they had taken in CS10 the previous week.

Some of my analyses consider individual participants while other analyses considers all participants from the pilot and first phase of data collection, all of whom answered the questions described in this chapter.

www.manaraa.com

Interview Protocol

During the interview, participants solved a series of problems and were asked to talk through their reasoning while they solved the problems. My protocol was modeled on diSessa's description of clinical interviewing (2007) and I provide details from my instantiation of these techniques here. From diSessa's description of clinical interviewing I have applied the following principles:

-

Before beginning the interview, the participant was provided and signed consent documents to participate in the research. The study was explained and any questions the participant had were answered. I explained to participants that I was interested in understanding how they thought about the problem and wanted them to talk aloud as they solved the problems. Participants in the pilot and the first phase of research who used the programming language Scheme were provided a warm-up question. Participants that used the programming language Scratch were not provided a warm-up question.

The intent of the warm-up question, shown in Figure 4, was for students to practice talking through their reasoning while solving a problem. After completing this question, I provided encouragement to the interviewees either to continue talking through their reasoning as they had done or to increase how much they were talking through their reasoning.

When solving the remaining problems, if a participant remained silent for an extended period of time, I prompted them to continue talking, for example by saying "what are you thinking?" If a participant asked a clarifying question about the problem, I redirected him or her to a relevant phrase within the text of the problem. If the participant provided an interpretation of the question and asked for confirmation, I provided confirmation if his or her interpretation was correct or redirected the participant to a relevant phrase within the provided question if his or her interpretation was not correct.

I attempted to avoid providing additional information to the participant or any indication to the participant regarding whether or not his or her answer was correct. For example, if a participant asked if an answer was correct, the interviewer frequently responded by redirecting the question back to the participant, for example by saying "What do you think?" The goal of this strategy was to provide additional insight into the participant's reasoning and to avoid providing additional resources such as whether or not their answer was correct.

After a student solved a problem, I frequently asked follow-up questions to attempt to better understand his or her reasoning. For example, occasionally I repeated back a statement the participant had said while solving the problem and asked what was meant by that statement. Similarly, I occasionally identified an element of a representation created by the participant and asked what that element meant or represented.

Occasionally these follow-up questions led the student to identify a mistake he or she made in solving the problem; however, the follow-up questions were intended only to clarify

www.manaraa.com

some aspect of the participant's reasoning that was perceived by the interviewer as ambiguous. These questions were not intended to serve as tutoring or to support the participants' reasoning. The student was not told if the answer to a question was correct or incorrect and was permitted to move onto the next problem regardless of whether he or she had answered the question correctly.

Analysis Methods

Analysis methods that were specific to an individual chapter are discussed within that chapter. The description of analysis methods described here are those that are applicable across the dissertation.

After each interview brief notes were taken regarding the content and quality of the interview. In particular I recorded my estimation of the quality of the interview for further analysis based upon the participants' relative comfort during the interview and the extent to which they were able to articulate their reasoning. These notes determined the order in which I viewed the videos, viewing first the videos of participants that appeared comfortable and were articulate about their reasoning. This may bias the results toward the reasoning of the more articulate participants because those interviews were watched first. However, in identifying case studies focusing on articulate students is necessary to provide the density of data necessary for careful analysis. All of the videos were watched and some of the analysis includes an analysis of all participants that answered particular questions.

After all data was collected from Phase 1, data analysis began by viewing these videos. This analysis continued during the collection of videos from Phase 2. Videos were watched from start to finish, pausing the video to take notes about relevant details and episodes. For example, a short summary of each participant's solution to each problem was recorded in addition to detailed notes regarding particular episodes. Episodes of interest were those in which a student appeared to be using a technique to reason about a problem where that technique was not specific to computer science. For example, I documented cases where participants used various forms of representation, used general test-taking strategies such as re-reading the question or working backwards from the answer options, rephrasing the question or program text in their own words, or performed calculations similar to mathematics calculations. In many cases the technique was not apparent and in others their technique seemed dominated by computer programming specific content knowledge. My records of these episodes of interest included the participant's identifier, the interview problem, the time within the interview, a short summary of the participant's behavior and reasoning, and the reason for my interest in this episode.

These episodes were documented on index cards and these index cards were sorted into clusters to attempt to identify relevant patterns within the data. Individual cases from these clusters were selected for further analysis. These cases were transcribed and descriptive memos were written for each to attempt to explain the content of that episode.

Additional details regarding the analysis methods are provided in the relevant analysis chapters.

www.manaraa.com

Recursion Background

The following section provides a detailed description of the focal interview questions and essential background information regarding recursion. Readers familiar with recursion may prefer to read the interview problems without the accompanying text that describes recursion and the problem solutions.

Two of the interview questions refer to a single recursive function and I will use this recursive function to introduce recursion in general. I will begin by describing the underlying recurrence relationship from the questions, shown in Equation 1. Equation 1 shows an equation relating exponentiation to repeated multiplication. For example, if the variable n is 3 and the variable x is 5, the equation in Equation 1 becomes $5^3 = 5*5^2$.

$$x^n = x * x^{n-1}$$

Equation 1. Recurrence relationship from the sample problem.

Equation 1 is a valid expression for representing an exponent to the power of one or higher. Equation 2 shows an expression for calculating the value of the variable x raised to the first power. This type of non-recursive expression in Figure 3 is typically referred to as a start condition in mathematics or a base case in computer programming (Leron & Zazkis, 1986).

$$x^1 = x$$

Equation 2. Base case from the sample problem.

Figure 2 shows a representation of the recurrence relationship from Equation 1 and base case from Equation 2, in the programming language Snap (Harvey & Mönig, 2010), which is a variant of Scratch (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008).

```
x to the n
script variables answer
if n = 1
    set answer to x
else
    set answer to x * (x to the (n - 1))
report answer
```

Figure 2. Implementation of recurrence relationship from Equation 1 and base case from Equation 2 in Snap

www.manaraa.com

The function shown in Figure 2 takes two arguments, which are set to the values of the variables x and n. For example, if we call the function with the arguments 4 and 2, the value of the variable x would be set to 4 and the value of the variable n would be set to 2. This function call would be made by double clicking on the expression shown in Figure 3.

Figure 3. A Snap function call provided the arguments 4 and 2.

If the value of the variable n is 1, the behavior of the program in Figure 2 is equivalent to calculating x to the first power (or x^1) as shown in Equation 2. The test for "if n equals 1" and the result of setting the answer to the value of x appears in the first half of Figure 2. For any value greater than 1, the variable named "answer" is set to the result of the algebraic expression $x*x^{n-1}$. This calculation is shown in the second line of the program in Figure 2 that begins "set answer." This requires multiplying the value of x, which is a known quantity, by the unknown quantity x^{n-1}. This unknown quantity can be determined by calling the function again. This is equivalent to using the mathematical representation shown in Equation 1 to calculate the value of x raised to the power of n. Then we use Equation 1 again to calculate the value of x raised to the power of n-1. In both uses of Equation 1, the value of n will be different, one less than the previous value of n.

This process of sequentially executing the function with lower values of n continues until the new value of the variable n is 1. At this point the variable named "answer" is set to the value of x and is returned to the previous function call. For example, the call to the function in Figure 2 with the arguments 4 and 2 would make a recursive call with the arguments 4 and 1. This function call with arguments 4 and 1 would return the value of x, 4, to the previous recursive call. In that previous recursive call, this returned value would replace the recursive call that was made there.

Interview Questions

Warm-up Question

The interviews in Phase 1 began with a warm-up question shown in Figure 4. Due to an omission in the preparation of materials, the two students who were enrolled in the Scratch-based programming course were not given a warm-up question.

What does `(mystery 3 10)` return?

```
(define (mystery x y)
     (+ 7 (* x 4) (* (/ y 5) (- x y))))
```

Figure 4. Warm-up question used during interviews in Scheme

After the interviewee answered the warm-up question, the interviewer provided the participant a stack of questions to answer with one question per page. The problems were multiple-choice format. Recall that these questions were the questions identified by Reges (2008) as those most highly correlated with success on the 1988 Advanced Placement Computer Science (APCS) exam translated into Snap and Scheme. These questions were chosen

www.manaraa.com

because they would align the interview with content from the APCS curriculum and because these correlations may indicate that the questions tested a core competence that was relevant across many multiple-choice and free-response questions on the exam.

Interview Question 1 – Tracing Question

A variant of the program shown in Figure 2, with an obfuscated function name "WhatIsIt", appeared on the 1988 Advanced Placement Computer Science (APCS) exam in the programming language Pascal. The interview participants were provided a version of this question translated into the programming language from their course. A version translated into Scheme is shown below in Figure 5. A version in Snap with the original function name is shown below in Figure 6.

The differences between these two representations of the same function may warrant curiosity regarding what differences in reasoning arise from these differences in programming language. In the data collected, there were no identifiable patterns of reasoning that separated participants who used each programming language. My hypothesis is that differences in participant reasoning caused by the programming language were insignificant compared to the variation between individuals. Given the lack of data, differences in representation will not be discussed further. For consistency and ease of reference, I will use the Scheme-based representation of functions and function calls for all inline references in the remainder of the dissertation.

What value is returned by `(WhatIsIt 4 4)`?

```
 (define (WhatIsIt x n)
     (if (= n 1)
          x
          (* x (WhatIsIt x (- n 1))))

A) 8   B) 16   C) 24   D) 64   E) 256
```

Figure 5. The "Tracing Question": a replication of a question from the 1988 APCS exam, translated to Scheme.

www.manaraa.com

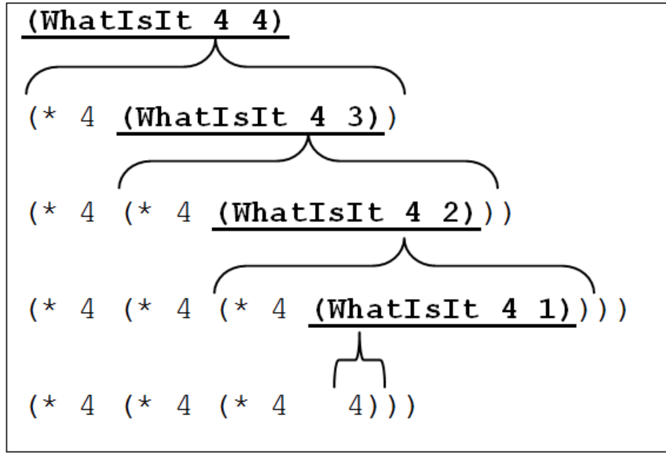
What value is returned by WhatIsIt 4 4 ?

```
WhatIsIt x n
if n = 1
    set return-value to x
else
    set return-value to x * WhatIsIt x (n - 1)
report return-value
```

A) 8   B) 16   C) 24   D) 64   E) 256

Figure 6. The "Tracing Question": a replication of a question from the 1988 APCS exam, translated to Snap.

The first question, after the warm-up question in Phase 1, asked students to calculate the value of `(WhatIsIt 4 4)`. Throughout the dissertation I refer to as the tracing question. The call to `(WhatIsIt 4 4)` generates a call to `(WhatIsIt 4 3)` and multiplies the result of that by 4. This process is repeated and the value of the variable x is repeatedly multiplied. The correct answer from this set of calculations is 256 or 4^4. Figure 7 shows the recursive calls generated by the initial call to `(WhatIsIt 4 4)`. The underlined portion on each line in Figure 7 is expanded in the next line to show the result of that recursive call. The final line shows the pending multiplications from the previous recursive calls and the value returned by the call to `(WhatIsIt 4 1)`.



www.manaraa.com

Figure 7. Recursive calls generated by a call to (WhatIsIt 4 4).

Interview Question 2 – Infinite-loop Question

The question shown in Figure 35 immediately followed the tracing question on the 1988 AP CS exam and was the second interview question used in Phase 1. This question, which I refer to as the "infinite-loop question," asked the student to reason about cases that do not create an infinite loop in the function `WhatIsIt`.

Which of the following is a necessary and sufficient condition for the function `WhatIsIt` to return a value if it is assumed that the values of n and x are small in magnitude and are both whole numbers?

A) $n > 0$
B) $n = 0$
C) $n > 0$ and $x > 0$
D) $x \leq n$ and $n > 0$
E) $n \leq x$ and $n > 0$

Figure 8. The "infinite-loop question": a replication of a question from the 1988 APCS exam.

The function `WhatIsIt` will terminate when the value of n is 1. However, if the value of n never becomes 1, a call to `WhatIsIt` will result in an infinite loop, meaning it will never terminate. For example, if the function `WhatIsIt` is called with a value of n less than 1, as shown in Figure 46, the recursive call will never stop.

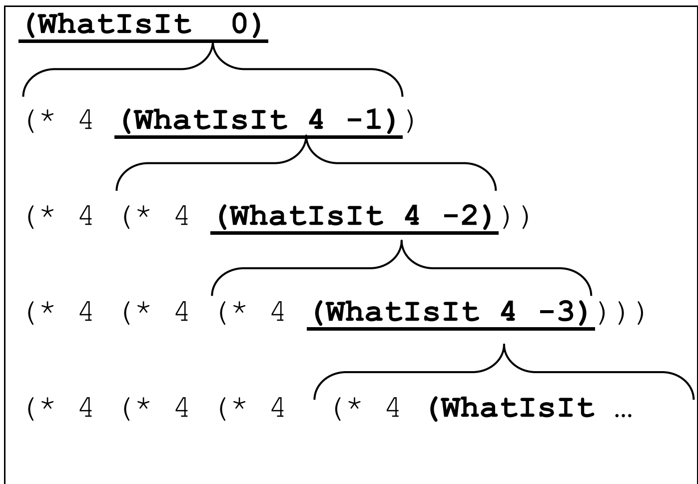


Figure 9. Recursive calls generated by a call to (WhatIsIt 4 0).

Interview Question 3 – Boolean Question

The third question asked students to select an answer option that described a line of code. Translated versions of the question are shown below in Figure 10 and Figure 11. This question was the question most highly correlated with success on the 1988 APCS exam (Reges, 2008).

If b is a Boolean variable, then the function below has what effect?

www.manaraa.com

```
(define (foo b)
    (let ((b (equal? b #f)))
    b))
```

A) It causes b to have value false regardless of its value just before the statement was executed.

B) It always changes the value of b.

C) It changes the value of b if and only if b had value true just before the statement was executed.

Figure 10. The Boolean question in Scheme, a translated version of the question from the 1988 AP CS exam

If b is a Boolean variable, then the statement below has what effect?

set b to b = false

A) It causes b to have value false regardless of its value just before the statement was executed.

B) It always changes the value of b.

C) It changes the value of b if and only if b had value true just before the statement was executed.

Figure 11. The Boolean question in Scratch, a translated version of the question from the 1988 AP CS exam

The Scheme equivalent of the question includes defines a function, using the syntax from the first line of Figure 10 "`(define (foo b)`." The argument to this function is "b," which establishes the variable named "b." This is necessary in the Scheme version because the participants from the Scheme-based courses did not yet have experience with persistent variables and only had experience using function arguments or "let" to create variables. Technically, by using "let," the Scheme version also creates a new variable named "b" rather than modifying the initial variable, but this is merely a limitation of using a functional programming paradigm.

Despite the differences, both versions of the code can be described in the same way. They both always change the value of the variable "b," which corresponds to multiple-choice option B. The expression tests if the initial value of the variable "b" is false. The result of this is set to be the new value of the variable "b." If the variable "b" starts out with the value of "true," then the test of whether it is equal to "false" will be "false" and the variable "b" will be changed from "true" to "false." If the variable "b" starts out with the value of "false," then the test of whether it is equal to "false" will be "true" and the variable "b" will be changed from "false" to "true."

www.manaraa.com

The original version of the question included two additional multiple-choice options. These options were both incorrect. These stated that "It causes a compile-time error message" and that "It causes a run-time error message." These multiple-choice options were not relevant for testing students using the programming language Scratch, where it is difficult to create syntactically invalid code. For consistency I removed both the Scratch and Scheme versions from the answer options presented to the participants regardless of what programming language they used during the interview.

Due to time constraints, participants' solutions to this problem are not described in detail in this dissertation.

Interview Question 4 – Wow Question

The fourth question asked students to calculate the value of `(Wow 16)`. The function `Wow` is shown below in Figure 12.

The procedure call `(wow 16)` will yield as output which of the following sequences of numbers?

```
(define (wow n)
    (begin
        (if (> n 1)
          (wow (/ n 2)))
        (show n)))
```

A) 10 8 6 4 2
B) 16 8 4 2 1
C) 1 2 4 8 16
D) 32 16 8 4 2
E) 2 4 8 16 32

Figure 12. The Wow question, a translated version of the question from the 1988 AP CS exam

This function included an "if" that does not have a false case. Most of the students misinterpreted the expression "(show n)" as a false case for the "if." This made it difficult to analyze participants' understanding of the recursion and therefore participants' solutions to this problem are not analyzed in this dissertation.

Interview Question 5 – Multiplication Question

The final question provides multiple-choice options to fill in the blanks in a function to recursively multiply two integers. The question shown in Figure 44 was provided to students using Scheme in their programming course and the question shown in Figure 14 was provided to students using Snap in their programming course. Each of these answer options specifies a distinct recursive function and below I explain the correct answer, option D, as well as the behavior of each of the incorrect answers.

www.manaraa.com

Consider the following function where

- x is an integer and x > 0
- It should calculate x * y

```
(define (mult x y)
   (if (= x 1)
       ;; statement 1
       ;; statement 2
   ))
```

Which of the following statement pairs properly completes the function?

```
      <Statement 1>              <Statement 2>
A.      (* x y)                  ;; none
B.      y                   (mult (- x 1) (+ y 1))
C.      y                   (mult (- x 1) (+ y y))
D.      y                   (+ y (mult (- x 1) y))
E.      y                   (* y (mult (- x 1) y))
```

Figure 13. Translated version of the multiplication question from the 1988 APCS exam in the programming language Scheme.

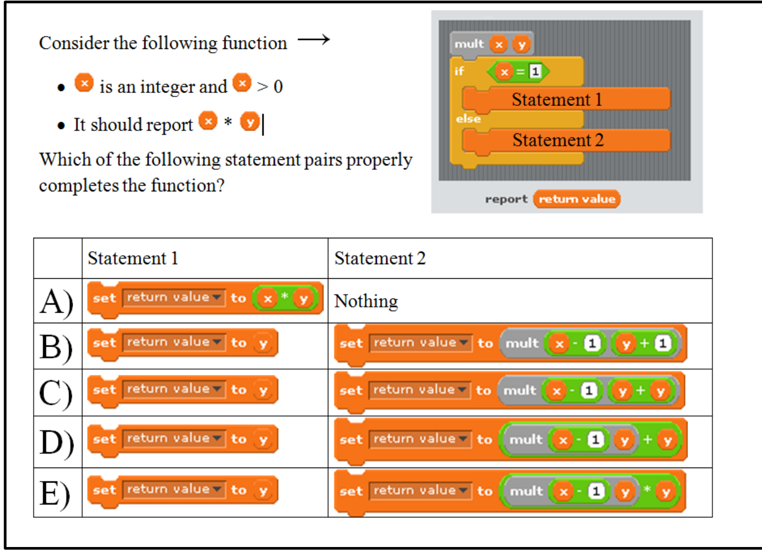


Figure 14. Translated version of the multiplication question from the 1988 APCS exam in the programming language Snap.

The question specified five options for completing the function `mult`; each option specified content for the true case for the "if" (statement 1) and the false case for the "if" (statement 2) for the function. For example, Figure 15 shows the correct completed function specified by answer option D.

www.manaraa.com

```
(define (mult x y)
     (if (= x 1)
          y
         (+ y (mult (- x 1) y))))
```

Figure 15. Correct version of the `Mult` function as specified by answer option D.

Answer Option A (Incorrect)

The recursive function specified by answer option A is shown in Figure 16 and is trivially incorrect. It does not specify a value for the false case of the “if” and therefore the `mult` function, as shown in Figure 16, only provides the correct answer when the value of x is 1. For all other values of x, the function does not multiply the values of the variables x and y.

```
(define (mult x y)
     (if (= x 1)
          x*y))
```

Figure 16. Incorrect version of the `mult` function as specified by answer option A.

Answer Option B (Incorrect)

Answer options B through E all provide the same value, y, for the base case, when x is equal to 1. This corresponds to multiplying the value of y by 1, which is always y. A mathematical representation of this base case for options B through E is shown in Equation 3.

$$1 * y = y$$

Equation 3. Based case specified by answer options B through E for the multiplication question from the 1988 AP CS exam

The function specified by answer option B is shown in Figure 17.

```
(define (mult x y)
     (if (= x 1)
          y
          (mult (- x 1) (+ y 1))))
```

Figure 17. Incorrect version of the `mult` function as specified by answer option B.

The recurrence relationship from answer option B is provided below in Equation 4.

$$x * y = (x - 1) * (y + 1)$$

Equation 4. Incorrect recurrence relationship as indicated by answer option B

www.manaraa.com

Equation 5 shows the recurrence relationship from answer option B expanded using algebra. From this expanded expression it is clear that x times y is not equal to the right hand side of the equation for all values of x and y and therefore that answer option B is incorrect.

$$x * y \neq (x * y) + x - y - 1$$

Equation 5. Expansion of incorrect recurrence relationship as indicated by answer option B

Figure 18 shows the resulting recursive calls from a call to `(Mult 4 4)` as specified by answer options B. The correct answer to return from this function call is the product of 4 and 4, 16. Between each call the value of x is decreased and the value of y is increased. When x is equal to 1, the value of y is returned. Therefore the final recursive call `(mult 1 7)` returns the value 7.

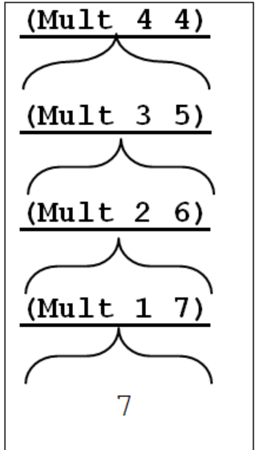


Figure 18. Recursive calls generated by a call to (Mult 4 4) with answer option B.

Answer Option C (Incorrect)

The function specified by answer option C is shown in Figure 19.

```
(define (mult x y)
     (if (= x 1)
          y
          (mult (- x 1) (+ y 1))))
```

Figure 19. Incorrect version of the `mult` function as specified by answer option C.

Equation 6 shows both the original recursive relationship specified by answer option C as well as a version expanded using algebra. The resulting calculation of 2y*(x-1) is clearly not equal to the product of x and y for all positive integer values of x and y.

$$x * y = (x - 1) * (y + y) = 2y * (x - 1)$$

www.manaraa.com

Equation 6. Recurrence relationship for representing multiplication as repeated addition as indicated by answer option C

Answer C follows a similar pattern to the recursive calls generated by answer option B. Figure 20 shows a function call of `(mult 4 4)` to the function specified by answer option C. The value of x decreases by 1 with each recursive call and the value of y is doubled with each recursive call. When x is equal to 1, the value of y is returned. Therefore the final recursive call `(mult 1 32)` returns the value 32, not the correct answer of 16.

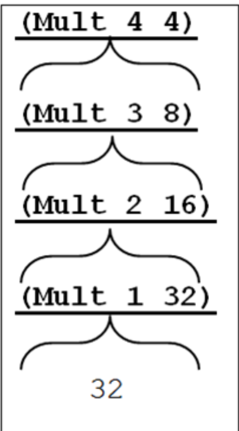


Figure 20. Recursive calls generated by a call to (Mult 4 4) with answer option C.

Answer Option D (Correct)

The function specified by answer option D is shown in Figure 21.

```
(define (mult x y)
     (if (= x 1)
           y
           (+ y (mult (- x 1) y))))
```

Figure 21. Correct version of the `mult` function as specified by answer option D.

Equation 7 is a mathematical representation of the recurrence relationship from the correct version of the multiplication function. This demonstrates the property that multiplication can be represented as repeated addition.

$$x * y = y + (x - 1) * \ y$$

Equation 7. Recurrence relationship for representing multiplication as repeated addition as indicated by answer option D

The value of x decreases at each recursive call, but the value of y remains the same. Figure 22 shows that each recursive call, shown underlined and bolded, can be expanded based upon the second statement in answer D. The statement `(+y (mult (- x 1) y))` adds 4, the value of y, to each successive recursive call. The final recursive call `(mult 1 4)` is 4, since the value of x results in it evaluating the base case. The base case returns the value of y to be

www.manaraa.com

combined with the previous pending sums. The diagram in Figure 22 shows how the repeated addition of 4 accumulates to result in the final calculation of 4+4+4+4.

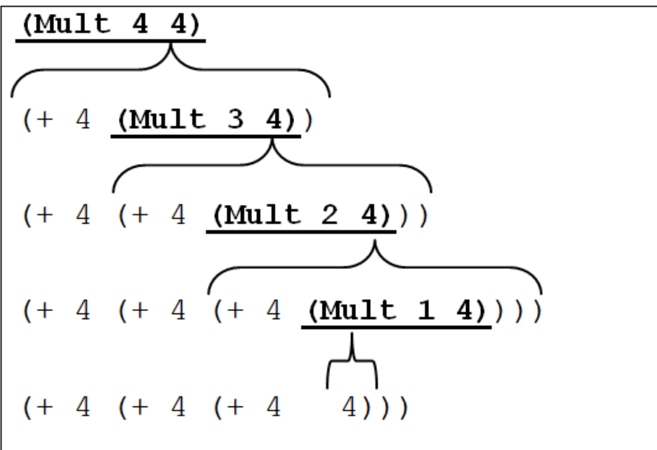


Figure 22. Recursive calls generated by a call to (Mult 4 4) with answer option D.

Answer Option E (Incorrect)

The function specified by answer option E is shown in Figure 23.

```
(define (mult x y)
     (if (= x 1)
          y
          (* y (mult (- x 1) y))))
```

Figure 23. Incorrect version of the `mult` function as specified by answer option E.

The recurrence relationship from answer option E, shown in Equation 8, is nearly identical to that of the correct answer. The only difference is that the value y is multiplied by rather than added to a recursive call to the function. Instead of multiplying the values x and n, this function multiplies x by itself n times. In other words, it calculates x to the power of n. Recall that the first interview question involving the function `WhatIsIt` performed the same calculation. The only differences between the functions specified by answer option E and the `WhatIsIt` function are the names of the variables and the name of the function.

$$x * y = y * (x - 1) * y$$

Equation 8. Recurrence relationship for representing multiplication as repeated addition as indicated by answer option D

Figure 24 shows a diagram parallel in structure to the one in Figure 22 for the function call `(mult 4 4)` for answer option E, which results in the value of 256 instead of the correct value of 16.

www.manaraa.com

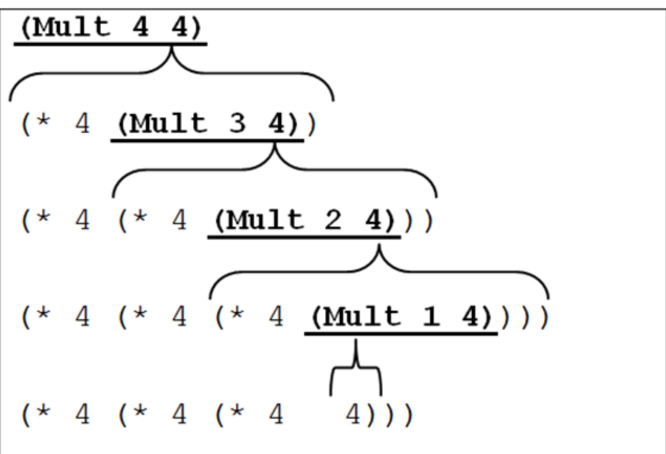


Figure 24. Recursive calls generated by a call to (Mult 4 4) with answer option D.

www.manaraa.com

THE COORDINATION CLASS OF STATE

> "State management is the essence of programming. Every technique and tool in the programmer's repertoire is concerned with supporting versatile and efficient management of the state space." (p. 6-7, Shinners-Kennedy, 2008)

Shinners-Kennedy argues that an understanding of state and state change operations is essential to successful programming and this argument has been made in many forms over the years. I define computer program state to include all values calculated and maintained by the machine when executing a program. This includes user-defined variables, arguments to functions, return values from expressions and sub-expressions, and stack information such as the program counter and nesting of function calls. du Boulay and his colleagues (du Boulay, O'Shea, & Monk, 1989; du Boulay, 1989) developed a claim similar to Shinners-Kennedy through a focus on what they call the *notional machine*, which is essentially a description of how program state can be inspected and changed. du Boulay and his colleagues argue that students need a firm understanding of these properties of the machine to be successful in writing programs and also that teachers and instructional materials should make these properties explicit to students. Twelve years later, Ben-Ari (2001) reiterated the importance of students' understanding of the notional machine and critiqued object-oriented programming languages for obscuring aspects of the machine. These ideas about the importance of state have come to fruition through the design of programming languages such as Logo, Boxer, Scratch, and Alice and one notable pedagogical approach (Sajaniemi & Kuittinen, 2005; Sajaniemi, Kuittinen, & Tikansalo, 2008); however, there are a number of open questions about the nature of students' knowledge about state.

A number of programming languages have been designed with the goal of making state visible (Papert 1980; diSessa, 2000; Cooper, Dann, & Pausch, 2000) The researchers involved in the design and evaluation of the Alice programming language emphasize the importance of program state, particularly for understanding and debugging code (Cooper, Dann, & Pausch, 2000; Dann *et al.,* 2003; Powers, Ecott, & Hirshfield, 2007). They assert that "the source of confusion in figuring out what went wrong, in all but the most trivial code, is an inadequate understanding of the program's state." (p. 109, Cooper, Dann, & Pausch, 2000). The design of the Alice programming language was informed by this emphasis on state; in the Alice programming language, commands can move the character's position on the screen, literally making state visible. This is the same mechanism of making state visible as was developed in the programming language Logo (Papert, 1980).

In response to the importance of state, Sajaniemi and Kuittinen (2005) have attempted to help students recognize common patterns of state change operations. Their pedagogy highlights the *roles* of variables in programs. They claim that 10 roles account for 99% of all variable roles used in introductory programming texts. In their more recent work (Sajaniemi, Kuittinen, & Tikansalo, 2008), they ask students to represent the state of an object-oriented

www.manaraa.com

program at a moment in time. They use what details a student represents as an indication of what aspects of state that student believe to be important. They track how what details a student represents changes during a programming course. These findings focus on how students represent the relationships between methods, classes, and objects in object-oriented programming.

While these researchers above argue for the pedagogical importance of program state and patterns in student learning, they do not investigate the nature of knowledge regarding programming state. Based upon the empirical work of Sajaniemi and Kuittinen (2005) and characterizations of the knowledge by du Boulay and his colleagues (du Boulay, O'Shea, & Monk, 1989; du Boulay, 1989), it appears that knowledge of state is assumed to be primarily factual in nature. I acknowledge that factual knowledge is important and possibly a prerequisite to competence with programming state, but I expect that expert knowledge of program state includes more than factual knowledge.

Previous research that explores the nature of programming knowledge moves beyond fact-centric models of program state knowledge, but does not attempt to explain the moment-by-moment interaction of knowledge. The first is diSessa's work with differences in structural and functional knowledge of computer programming (1986). The second is the work of the BRACElet project, which has investigated a possible hierarchy of programming skills (Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009).

diSessa (1986) discusses two complementary models of individuals' understandings of a programming environment. An individual can have elements of a structural model, which, like the idea of a notional machine (du Boulay, O'Shea, & Monk, 1989; du Boulay, 1989), is a precise model for the state change operations in the system. An individual can also have elements of a functional model, composed of particular ways of accomplishing things in the programming environment. diSessa (1986) highlights the need for mutual bootstrapping between these two types of models. A structural model alone provides a barrier to early learning and is unlikely to support "fluid interaction with the system" (p. 205, diSessa, 1986). A functional model may support fluid interaction, but alone does not provide the necessary knowledge for debugging a program line by line. While facts are necessary for both a structural and functional model the distinction between functional and structural knowledge is independent of whether this knowledge can be classified as factual.

Researchers on the BRACElet project have set out to achieve a similar goal of meaningfully segmenting programming knowledge (Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009). In contrast to diSessa's model of a mutual bootstrapping process between different types of knowledge (1986), these researchers claim that there exists a hierarchy of programming skills. The methods employed in the BRACElet project are primarily quantitative, based upon students' responses to code writing, tracing, and explaining tasks. They identify knowledge of basic programming constructs as at the bottom of the hierarchy and writing code at the top.

www.manaraa.com

The BRACElet project researchers draw these conclusions based upon patterns of students' responses to the questions that test these competences, and their data suggest the existence of a hierarchical dependency. However, the same patterns would be present if the questions testing skills at the top of this hierarchy were coincidentally the most difficult. Existing models have made important theoretical contributions to understanding the domain, but do not provide the granularity to explain moment-by-moment dynamics of individuals' reasoning about program state.

These models are the state of the art in computer science education, but models of knowledge in other domains are more ambitious regarding moment-by-moment analysis of individuals' reasoning (*e.g.*, diSessa, 1993; diSessa & Sherin 1998)

To contribute to our understanding of the nature of computer programming knowledge, I show how a formal model of a particular type of concept applies to the learning of computer programming. Previous research from physics education (diSessa & Sherin, 1989) developed a theoretical model for a particular type of concept that the authors refer to as "coordination classes" (diSessa & Sherin, 1989). The details of this theoretical model were described in the theoretical framework section. This model stipulates that a coordination class is used by individuals to identify some focal information in the world. For example, in the case of the coordination class of force, this requires identifying forces, which includes identifying the position, direction, and magnitude of the force. Coordination classes have been identified within physics (diSessa and Sherin, 1998; Wittman, 2001; Parnafes, 2007; Levrini & diSessa 2008) and mathematics (diSessa & Wagner, 2005; Wagner, 2006). It is an empirical and theoretical question whether coordination class theory applies to concepts outside of these domains, but this theoretical model provides the potential to move toward moment-by-moment models of students' understanding of the importanct concept of state. Drawing from a larger study to be described later, in the following analysis I argue that the theory is relevant to computer science and can be refined by application to the concept of state.

In the current analysis I propose that the concept of state is a coordination class. In the computer science context, state includes values calculated and maintained by the machine when executing a program. This includes user-defined variables, arguments to functions, return values from expressions and sub-expressions, and stack information such as the program counter and nesting of function calls. I show that students use everyday knowledge when reasoning about computer program state.

This coordination class analysis requires command of the possibly unfamiliar vocabulary provided in the theoretical framework chapter. I believe that using coordination class theory provides other benefits, which in this analysis outweigh the costs. For example, the application of coordination class theory in this analysis was motivated by the following theoretical and pedagogical goals.

A first theoretical goal was to use precisely defined terms to describe the nature of computer science knowledge. The coordination class constructs serve to name and operationalize aspects of the participants' knowledge, reasoning, and performance that are

www.manaraa.com

relevant to understanding their solution paths. In the process of describing rich episodes from the data corpus, it is beneficial to utilize established constructs. This is a better alternative than inventing new constructs because it likely provides additional clarity by using time-tested and validated constructs and increases comparability to previous studies.

The second theoretical motivation for this work was to extend coordination class theory outside of physics and mathematics. Coordination class theory is not expected to be a static fully-refined theory (Cobb, Confrey, diSessa, Lehrer, & Schauble, 2003). Instead, it is expected to be refined and tested by additional researchers. For example, many researchers have introduced new constructs and engaged in other forms of theory refinement (diSessa & Sherin, 1998; Wittman, 2001; diSessa & Wagner, 2005; Wagner, 2006; Thaden-Koch, Dufrense & Mestre, 2006; Parnafes, 2007; Levrini & diSessa 2008). My analysis provides the first analysis of a coordination class outside of the domain of physics or mathematics. The successful application of the theory to this new domain provides additional validation of the theory and this extension also works toward identifying commonalities and differences in learning across domains.

The third theoretical goal was to make coordination class theory more comprehensible and therefore more valuable to the educational research community. diSessa and Sherin (1998) explicitly used empirical data as a tool to make the constructs from coordination class theory better understood by readers. Along the same lines, a goal of my empirical analysis is to contribute an example that can make coordination class theory more comprehensible, particularly for computer science educators.

The first pedagogical goal is to develop better models of the commonalities and differences in learning across domains. This has immense potential for computer science education, which is a relatively young field compared to mathematics or physics. These non-computing domains have a much longer history of educational research and may provide insights regarding teaching and learning that could improve computer science education. A missing link in connecting these bodies of research is the open questions regarding the commonalities and differences in learning between computer science and other domains.

The second pedagogical goal is to join others (Papert, 1980; diSessa, 1986; du Boulay, 1989; Cooper, Dann, & Pausch, 2000) in identifying state as a central concept in computer programming. I hypothesize that there may be commonalities in students' knowledge of state across everyday and computer science contexts. Focusing students' attention on state and their relevant prior experience with state may help students be more effective in learning about new types of state.

As a first pass, state is a candidate for a *coordination class* because an expert in dealing with state can work with state fluidly across contexts to identify a type of information (diSessa & Sherin, 1998) from the world or, in this study, to identify state in computer programs. Coordination class theory provides a target for students' understanding of computer program state. If state is a coordination class, competence with computer programming state includes not only the necessary facts regarding the behavior of the programming language, but the

www.manaraa.com

coordination of this knowledge to achieve consistent and correct performance within various contexts (referred to as *alignment* by diSessa & Sherin, 1998). To name only a few considerations in this process, individuals must use knowledge about the scope of variables and about state change operations, which includes control structures that change the state of the program counter. The full set of knowledge, including knowledge related to these components and others, which an individual can use to identify the focal information of the coordination class, is referred to as the individual's causal net (diSessa & Sherin, 1998).

The focus on state rather than computer program state or a sub-component of computer program state such as arguments, variables, or call-stack state is a decision with theoretical and practical implications.

Previous coordination class researchers have discussed the difficulty of identifying what coordination class individuals are using because many coordination classes are highly interconnected (Thaden-Koch, Dufrense & Mestre, 2006). diSessa and Wagner (2005) developed the idea that coordination classes exist in close relations to a collection of other coordination classes called *coordination clusters*. For example, diSessa & Sherin (1998) primarily emphasize force, which operates in a similar cluster of closely related coordination classes such as position, velocity, and acceleration. It is likely that no coordination class exists in complete isolation from other coordination classes, but coordination class researchers typically still discuss the role of a single coordination class (Wagner, 2005; Wagner, 2006; Levrini & diSessa 2008) or a few coordination classes (diSessa & Sherin, 1998; Wittman, 2001; diSessa & Sherin, 1998; Parnafes, 2007) as primary in students' reasoning.

I chose to analyze the coordination class of state. An alternative would be to analyze individual components of state separately. I hypothesize that many of the subcomponents of state have significant conceptual overlap and that identifying these commonalities may be beneficial for students and educators. For example, I expect that the competence required identifying the state of variables overlaps significantly with the competence required to identify the state of user-defined variables. This is not a hypothesis that I systematically validate in this study, but it served to motivate the selection of the focus on the unified coordination class of state.

I focus on the coordination class of state rather than only computer program state because I build upon the hypothesis from Knowledge in Pieces (diSessa, 1993) that domain expertise may develop from and include intuitive knowledge. Previous research in the Knowledge in Pieces line has focused extensively on the use and productivity of students' intuitive knowledge in physics (*e.g.*, diSessa, 1993; diSessa &; Parnafes, 2007). Everyday interaction with the physical world provides an obvious source of intuitive knowledge. In programming, the existence of relevant intuitive knowledge is less obvious, but potentially just as rich. Nevertheless, if program state is a coordination class, as I propose, then intuitive knowledge should interact with and in some cases support expert knowledge.

Shinners-Kennedy (2008) argues for the "everydayness" of state. He enumerates a diverse set of stateful systems that permeate everyday life. From a wedding band indicating an

www.manaraa.com

individual's state as married or single to a scoreboard showing the score in a sporting event, he argues that people are essentially experts in reasoning about state. For example, the score in a sporting event changes with a specific set of known state-change operations. Shinners-Kennedy's arguments align with my expectations of individuals' experience with state. State as a concept is not limited to a programming context. As Shinners-Kennedy argues, stateful systems and representations of those states surround us. While the relevance of this experience to computer programming is contestable, the existence of these stateful systems and therefore individuals' experience with them is not. Given this ubiquitous experience with stateful systems and representations of those states, it may be reasonable to expect that individuals have developed intuitions regarding stateful systems. With this potential source of rich intuitive knowledge regarding everyday examples of state, it is relevant to question how this knowledge of state might be used in the development of expertise with computer programming state.

The case study in this chapter is taken from a student reasoning about a computer science problem, and her everyday knowledge of state is of central focus in understanding her reasoning.

diSessa and Sherin (1998) provided theoretical and empirical requirements for identifying a coordination class. This section provides an argument regarding the plausibility that state is a coordination class. While this mapping between the requirements of coordination class theory and state present a plausibility argument that state is a coordination class, it is possible that the behavior of individuals' knowledge would not match the predictions based upon the model of the nature and interaction of knowledge. Therefore it is important to provide empirical validation of the model. This is not done using a large-scale quantitative study, but instead using process data from individuals' reasoning about relevant problems.

The data are taken from a portion of an interview with a college student named Megan (pseudonym) while she was solving a question taken from her previous course exam. During the focal episodes, she considered the possible values of two variables "A" and "B" and the behavior of the conditional "and" in the expression "A and B." From the computer science perspective she was attempting to identify the set of legal states of the variables "A" and "B" and the output state of the "and." During the focal episodes Megan used her everyday knowledge of both "if" and "and" to reason about the expression "A and B."

I will present four episodes. The four episodes are sequential and show various components of her coordination of state for the conditional "and." These episodes document challenges and opportunities to building upon everyday knowledge. In each of the excerpts I identify active components of her coordination class of state such as elements from her causal net that are and are not used in concept projections to determine the behavior of "and" for different input states.

Methods

The episodes presented in this analysis are taken from a larger study designed to identify students' productive out-of-domain knowledge. The data consists of videotapes of

www.manaraa.com

semi-structured clinical interviews. The design of the interviews and study was informed by a line of work focused on the role of students' prior knowledge. This line of research, referred to as Knowledge in Pieces (diSessa, 1993), provided me methodological examples (diSessa & Sherin, 1998; Wittman, 2001; diSessa & Wagner, 2005; Wagner, 2006; Thaden-Koch, Dufrense & Mestre, 2006; Parnafes, 2007; Levrini & diSessa 2008) and is the line of work from which I draw coordination class theory. The hypothesis that program state was a coordination class had been developed prior to data collection for this study.

The participants were recruited from two introductory programming courses at the University of California, Berkeley and the interview used the programming language from the participants' course, either Scheme or a variant of the Scratch programming language (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) known as Snap (Harvey & Mönig, 2010).

Content logs were created of all video data and episodes in which a participant experienced difficulty reasoning about or tracking state were flagged for further analysis. The episodes were of interest because when a participant experienced difficulty solving a problem he or she would frequently make statements that made aspects of his or her reasoning visible. Episodes of an individual solving a problem correctly often did not include evidence of that individual's reasoning process. To evaluate theories of learning and knowing it is necessary to select episodes that provide sufficient information regarding the participant's reasoning (diSessa, 1993). I flagged examples when an individual experienced difficulty because of the prevalence of elaboration of an individual's reasoning in these situations. These excerpts were transcribed and annotated with information regarding gestures and inscriptions made by the participant.

Ultimately, the episodes presented in this case study were selected because of Megan's articulateness about her reasoning and because they provided the opportunity to discuss a number of phenomena described in previous coordination class analyses.

The presentation of data is separated from the coordination class analysis of these data. For each episode I provide a summary of the episode, which is a narration of the episode and attempts to provide a clear representation of Megan's reasoning and her interactions with me, the interviewer. Extended quotations are provided in the summary to provide the reader the opportunity to evaluate the validity of the analysis (Corbin & Strauss, 2008). Portions of the interview that are not of central interest are summarized in lieu of extended quotations.

Following each summary section, I interpret Megan's knowledge and reasoning using coordination class analysis and my interpretations are drawn from these extended quotations. I identify specific causal net elements that Megan appeared to use to identify state. In comparison to analysis of students' understanding of physical phenomenon (diSessa & Sherin, 1998) these are only a best approximation of the nature and scope of this knowledge because there is no previous research in computer science documenting knowledge elements as has been done in physics (diSessa, 1993).

www.manaraa.com

Explanation of Focal Question

The following section provides background regarding the interview context and details regarding the programming content for readers less familiar with computer programming or the programming language Scratch.

During the interview, Megan solved a problem from a recent exam in her introductory programming course, which asked what the domain and range were for the function foo shown in Figure 25. The episodes in this case study focus on her reasoning about a subset of this problem.

Figure 25. Original problem context from Megan's exam

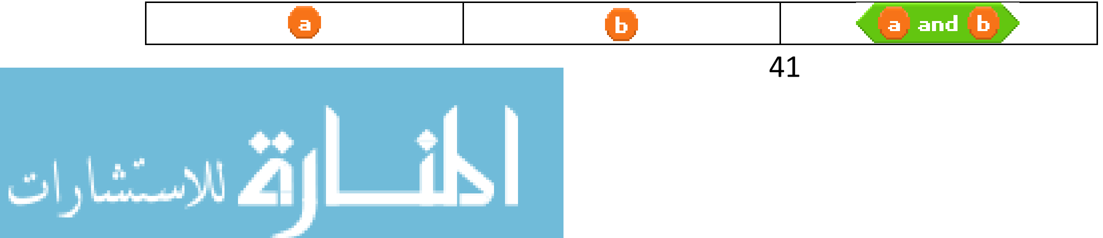
When solving this problem, she discussed at length the expression “A and B,” shown in Figure 26, and these discussions are the focus of the analysis. Figure 26 includes the variables a and b. I will refer to these variables as “A” and “B” to clearly distinguish the variable a from the article: a. Relating to the original problem context, Megan reasoned about what the variables “A” and “B” could be.

Figure 26. Focal expression considered by Megan.

As background, variables in the Snap programming language, like many programming languages, can be set to a value of true or false. However, students may only be exposed to a more narrow set of possible states of variables, not including Booleans. Sajaniemi (2002) found that of the 557 variables provided in three introductory programing textbooks, only six (or about 1%) of the variables stored a Boolean value. This suggests that students might have little exposure to cases like the one shown in Figure 26, where the variables “A” and “B” store a Boolean value. However, Booleans are not generally unfamiliar to students. Novice programmers commonly use Booleans in conditional control structure such as “if.” It is an open question whether students who do not find using Booleans in control structures to be challenging do find storing Booleans in variables to be challenging. A related observation is that novices will frequently avoid a return statement that returns a Boolean without enclosing the return statement within an “if.” Clancy (2004) reports that students will rewrite a statement like “return x = = y;” as “if (x = = y) return true; else return false;”

The conditional “and” as shown in Figure 26 works like a function that starts with the initial state of two arguments and then returns a value. The conditional “and” will return true if both of the arguments are true and otherwise will return false. This output state from “and” may be stored in a variable or be the input to another function. Figure 27 shows a summary of the behavior of the conditional “and.”

a	b	a and b



www.manaraa.com

True	True	True
True	False	False
False	True	False
False	False	False

Figure 27. Truth table for the state of the variables “A” and “B” and the resulting output of the focal expression “A and B”

In many programming languages, the equivalent of the expression from Figure 26 is not valid without additional information specifying what to do with the return value from the “and.” Traditionally, the return value would be used in a control structure such as “if” or “while.” In Snap, the language used by Megan, it is possible to execute this expression without additional code; you can simply double click on any expression and a small speech bubble shows the return value.

The “if” control structure can be thought of as a function that takes two arguments: a Boolean and an expression or set of expressions. These expressions are executed if and only if the Boolean provided to the “if” was true. The “if” is not responsible for the processing of the conditional expression, only operating on that intermediate state that is returned by the conditional expression. Consider the expression shown below in Figure 28.

Figure 28. Example "if" expression in the programming language Scratch.

Students may incorrectly believe that the “if” operates on an expression such as “n = 1.” Instead it takes the result of evaluating that expression, which will be either true or false. This may make the “if” appear less “smart” than might otherwise be assumed by students (see Clancy, 2004). A relevant element of structural knowledge, (diSessa, 1986) is what state is accessible to the “if.” The “if” only receives as input the result or state that is output by the expression contained as the test in an “if” expression.

Case Study: Megan

This case study is broken up into four episodes that were divided at points where Megan noticeably changed the causal net elements in her concept projection of state or her reasoning about state. These changes were also accompanied by differences in her coordination of program state. The narration and the coordination class analysis of each episode are presented sequentially below.

Episode 1

Summary of Episode 1

As previously mentioned, the expression considered in this set of episodes was the expression “A and B.” This episode began when I asked Megan, *“Yeah how does ‘and’ even*

www.manaraa.com

work? Do you know what I mean? Like let's say if you were explaining it to somebody." Megan's response to this question is shown below.

> *"Um, I've used 'and' just to like combine two things, so saying like ok if my shirt is red and I'm wearing shoes then this is true, like I'm matching or whatever. (Interviewer: Yeah) So then the 'and' would be used to combine two things."*

Megan then returned to the original issue of what the variables "A" and "B" could be. She concluded that *"they can't be numbers"* and then correctly summarized the behavior of the "and" expression. *"if 'A' is true and 'B' is true then the Boolean is true. I think. And then. Oh so if only one of them is true then the whole thing is going to report false, because 'and' means both of them."* She again returned to this discussion of what values the variables "A" and "B" could be and brainstormed possibilities other than numbers.

After this digression, Megan went on to reiterate the cases in which the expression "A and B" will return true or false. Although I interpret her tone and pacing as uncertain, her reasoning was accurate for all cases. Table 1 shows the transcript for Megan's unprompted explanation of the various cases. The transcript is paired with the truth table for the expression "A and B," and the ordering of these quotations and cases corresponds with the original ordering of her statements.

Table 1. Transcript of Megan's description of the cases of the expression "A and B," shown alongside the relevant line of the "A and B" truth table.

Transcript	a	b	a and b
"it would only report true if both are true."	True	True	True
"If one of them was true and one of them was false what would it report? (Interviewer: Yeah, what do you think?) It should report false then."	True	False	False
	False	True	False
"if both are false then it also would report false? (Interviewer: What do you think?) I, I think it would report false."	False	False	False

Analysis of Episode 1

Although Megan's explanations of "and" being used to *"combine two things"* and that *"'and' means both"* are far from what might be found in a computer science textbook, her performance in describing the behavior of "and" in each of these cases demonstrated appropriate coordination. I will identify some elements of her causal net from her concept projections in this episode and I will discuss how these causal net elements supported her inferences and coordination of state for each of the three cases she discussed.

www.manaraa.com

This excerpt began with Megan responding to the question of *"how does 'and' even work?"* Her statement *"if my shirt is red and I'm wearing shoes"* makes no reference to a computer science context and from this I assume she is using an everyday and not a technical use of the word "and." This statement, which used a non-computer science use of "and" and a clothing context, suggests that she was building upon her everyday knowledge of "and." However, an individual's everyday knowledge of "and" is likely varied. From her use of a non-computer science version of "and" I assume that she is using her everyday knowledge of "and," but from this alone cannot identify particular elements in her causal net.

Megan's initial explanation of "and" is focused on the way in which "and" works to *"combine two things."* The idea that "and" works to *"combine"* is a relevant element of Megan's causal net for identifying state. The word *"combine"* is not specific enough to be correct or incorrect in the computer context and can be generously interpreted as a summary of the fact that there are two inputs provided to "and," which are in fact logically combined[1]. It is ambiguous if this causal net element, which I will refer to as *"and" works to combine,* is used in her concept projection for each of the cases in Table 1, but this causal net element is consistent with her inferences in each of these cases.

After she described the behavior of "and" Megan justified her conclusions by saying *"because 'and' means both of them."* I do not anticipate that individuals will necessarily be able to identify the rationale behind their reasoning (diSessa, 1993); however, here Megan makes an explicit connection between her conclusions about "and" and this idea. I interpret her statement *"'and' means both of them"* as indicating generally that both inputs to the "and" must be true for it to return true. However it is unclear what properties Megan assumes "both of them" have. In particular, it is not clear if Megan realizes that the inputs to the "and" are Booleans and not unevaluated expressions.

This phrase, *"'and' means both of them,"* can serve as a rule for determining the behavior of "and." While one person might directly memorize the content of the truth table for the function "and," the idea that *"'and' means both of them"* can be used to derive the truth table. This is a second element of her causal net that supported her determination of the behavior of "and" and I will refer to it as *"and" means both of them*. I use Megan's language to describe this causal net element, but I do not assume the form of the knowledge is only linguistic. What Megan appeared to achieve here was the determination of the behavior of "and" through the application of this idea to three cases. That determination of the behavior of "and," if derived from the idea that "'and' means both of them" is not best described as linguistic knowledge.

Megan's statements shown in Table 1 are correct. They are also consistent with, and possibly derived from, from her causal net element that *"'and' means both of them."* Megan appeared most confident when she claimed that *"it would only report true if both are true."* Her language of *"means both of them"* can be mapped to this case where she describes that *"both*

[1] The language of "input" is not used by Megan, but for clarity is used in my explanation here and in the following paragraph.

www.manaraa.com

are true" using the common language of "both." The simplicity of this mapping may account for her relative confidence in the behavior of "and" for this case. This may be a first example of applying this causal net element to make an inference about the behavior of "and." Coordinating state for the remaining cases, she may have continued to apply her causal net element that *"and" means both of them*, but in these later cases Megan appeared less confident. Her confidence may relate to the relative difficulty of this mapping her causal net element *"and" means both of them* to these later cases. Megan's lack of confidence can be seen in her interactions with me during the interview where she frequently phrased conclusions in the form of a question. Megan seemed unsure of what the expression would return if provided one true value and one false value; however, she identified the correct answer without assistance from me: *"If one of them was true and one of them was false what would it report? (Interviewer: Yeah, what do you think?) It should report false then."* Considering the final case, Megan stated that: *"if both are false then it also would report false?"* I responded by saying *"what do you think?"* and Megan with some hesitation said *"I, I think it would report false."*

Despite some uncertainty throughout her explanation, I see no evidence of flaws in Megan's reasoning. From the perspective of coordination class theory, she used elements from her causal net in concept projections to draw correct inferences about each of the cases. She demonstrated appropriate coordination of program state in the context of "and." In particular, it appears that Megan could have used her causal net elements that *"and" works to combine* and that *"and" means both of them* to make inferences about the behavior of "and."

While she deliberated when identifying the output state of the "and," she seemingly without thought identified three relevant cases for "and" as two true values, two false values, and one true and one false. I believe she was making inferences regarding the output states for "and" using the two previously mentioned elements of her causal net. However, to produce these cases she likely had a relevant causal net element, or elements, to more directly determine the relevant cases. Unfortunately we do not have data regarding the possible elements that supported this quick and accurate identification of cases, but according to coordination class theory whatever knowledge supported this aspect of her reasoning would count as an element of her concept projection for each of the cases.

While the causal net elements discussed thus far are likely derived from everyday knowledge, her language of "report" is suggestive of a computer science context because the term is used in the Scratch programming environment. It is unclear if she is conscious of whether she is using computer science or everyday knowledge. I interpret Megan's statements as connecting her everyday knowledge that *"and" works to combine* and that *"and" means both of them* to a computer science version of "and," which "reports" a value. This bridging of knowledge may account for Megan's uncertainly, or her uncertainty may come in attempting to apply a rather ambiguous rule such as "'and' means both of them" to predict the behavior of "and" in the computer science world.

www.manaraa.com

My central claim in this episode is that Megan is reasoning about "and" in an everyday usage rather than only a computer science usage. For example, the "and" appearing in "rice and beans" is distinct from the computer science version of "and," but could appropriately be described as *"combining two things"* and is consistent with *"'and' means both of them."* This everyday "and" in "rice and beans" does not include the complexity of the computer science "and" where it receives as input only Boolean values and does not have access to the larger expressions that may be in an "and" expression. Her explanations of "and" do not suggest that they are computer-science-specific explanations of "and," but we can see how these causal net elements that I identified could be used in a concept projection to produce coordination of program state.

Her understanding of the context she created relating to "*if wearing a red shirt"* and *"if wearing red shoes,"* is undeniably supported by out-of-domain knowledge. I believe that she is able to cue her knowledge regarding conditionals in English and use it to understand the expression "true and true." In this context, she demonstrated proper coordination and reasoned that *"if 'A' is true and 'B' is true, then this would be true."*

Episode 2

Summary of Episode 2

Within less than a minute of the previous episode, Megan expressed a different set of ideas regarding the expression "A and B." In the interim time, Megan responded to my question regarding whether or not she thought she could use Scratch to confirm the conclusions she had made. A transcript of this digression is not provided. Megan's statements that are the focus of episode 2 are shown below. In this segment, Megan began by questioning her previous reasoning.

> *"I don't understand how this would work because if you just have true and true why would that report true? Like shouldn't, then again it could. (pause) Ok if we set it, but if you're setting it, so like the Boolean of true and false, I still don't get like why is that going to report anything."*

To try to understand her seemingly new set of ideas I asked her "c*an you draw that in Scratch, what like, what you mean? The Boolean of true and false."* She spoke as she created the first three rows of inscriptions shown on the left side of Figure 29, but this transcript is not provided. The left side of Figure 29 shows the full representation she completed during the following quote. The right side of Figure 29 shows a translated representation of the expressions she had written, shown in the programing language Snap.

www.manaraa.com

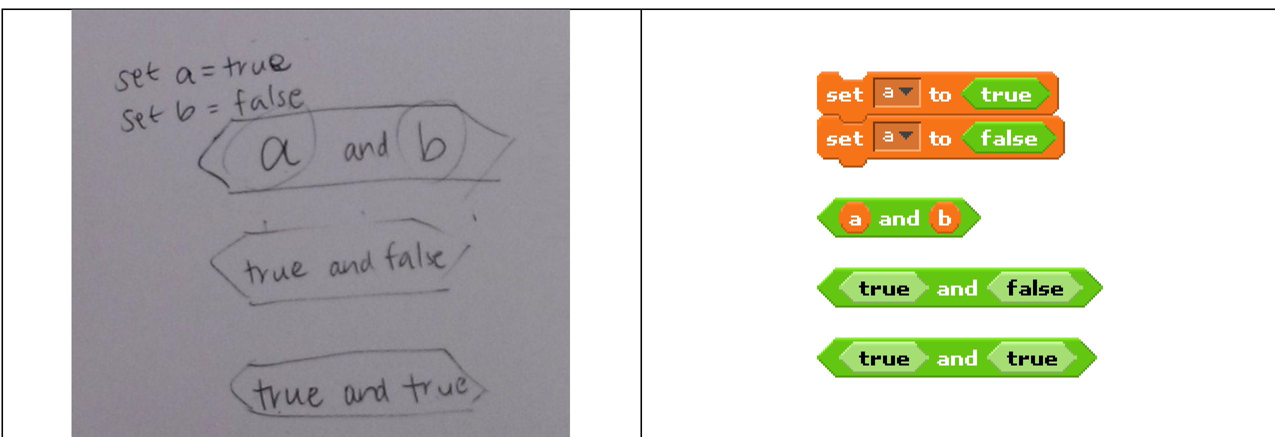


Figure 29. The work written by Megan when considering behavior of the conditional "and" with arguments "A" and "B."

She made the following conclusions:

"Then you would basically have like the Boolean of true and false (writes "true and false"). But that's not going to report anything because why would it report something that, is this true? Well is true and false true? Not really. And is, it's not false either; I don't think that could be a possible input. (Interviewer: And what part makes it impossible?) Just that it doesn't make sense, to have like even if you have both of them true, it's just why would it be true and true? (Writes "true and true") True and true make what? Like, that's what I don't get. Why would it make true?"

Megan's next statements provided insight into her different inferences in episodes 1 and 2. Megan distinguished two contexts of consideration: *"out here in this world"* versus *"the computer"* and determined that different rules applied in each context.

"I mean it makes sense like out here in this world for it to be true just because it's true and true, but I don't see why the computer would make it, this equal true."

Analysis of Episode 2

In episode 1 Megan demonstrated no flaws in reasoning about the expression "A and B." In episode 2 she did not demonstrate the same competence and questioned her previous conclusions regarding the behavior of the conditional "and." Megan's final statements of this episode provided an explanation for her change in reasoning. Essentially she said that her everyday assumptions about "and" were not necessarily applicable to predicting what the computer would do.

Regarding the expression "true and false," Megan said that *"I don't think that could be a possible input."* This may relate to a lack of knowledge that an "and" is provided two Boolean values and not two expressions. Megan may or may not have this computer-science-specific knowledge in her causal net, but she does not apply it in this context.

She confidently claimed that "true and true" *"makes sense like out here in this world."* She appears to find "and" *"out here in this world"* intuitive and barely requiring an explanation.

www.manaraa.com

To justify why *"true and true"* would be true *"in this world,"* she provides only a minimal explanation. She said "*I mean it makes sense like out here in this world for it to be true just because it's true and true*," which amounts to little more than saying *"just because."*

This shows that Megan recognized that we can think about "and" in an everyday context or a computer context. Her belief that there may be different behaviors in these two contexts is accurate. This is a primary causal net element that dictated important aspects of her reasoning in this episode, which I refer to as *this world and the computer have different rules*. Megan does not make use of her causal net elements that *"and" works to combine* and that *"and" means both of them*.

From episode 1 we have evidence that she had causal net elements to generate appropriate coordination, but here she is not successful at identifying the behavior of "and." In fact, this is an issue of span because she does not believe her knowledge to be relevant to this context. It is not an external change in context, but Megan considers the context in a new way, which prevents her from drawing inferences and creating a concept projection for any of the cases.

Recall that a concept projection is composed of all causal net elements that guide extractions and that are used to generate an inferential chain to identify the focal information of that coordination class. Megan uses the causal net element that *this world and the computer have different rules*. According to coordination class theory, Megan does not generate an inferential chain in this context because this causal net element causes Megan to believe her knowledge is not relevant. Unlike the other causal net elements I described, this does not get used within an inferential chain, nor does it guide an extraction. It is part of Megan's causal net that is activated in this context, but there is no inferential chain and therefore no concept projection for this to be a part of.

Megan does not continue to use the computer science laden word "report" here, which I identified as playing a role of bridging her everyday and computer science knowledge. Instead of using the technical word "report," Megan used the word "make." She said "True and true make what? Like, that's what I don't get. Why would it make true?" In episode 1 Megan used the technical language of report when using primarily everyday knowledge of "and." However, in episode 2 when she explicitly discussed the computer context, as in the statement "I don't see why the computer world make it, this equal true," she used the everyday and non-technical language of "make."

It is unclear what caused Megan's shift in reasoning and her new attention to the context of "and." My prompt to *"draw that in Scratch"* brought the computer science context to her attention, but this prompt was in response to her change in reasoning and cannot be seen as the cause of this change. While we can rule out this prompt as the cause of the change in her reasoning, a similar mechanism might have played a role. During the interview, a copy of her test, with the expression "A and B" was visible. While there are no gestures that showed her attention on the test, seeing the test could have cued her attention to the computer science context and provided a greater attention to context in general.

www.manaraa.com

The primary observation in this episode is that Megan engaged in a change of reasoning based upon her causal net element that *this world and the computer have different rules*. This change in what causal elements were used caused a lack of span because Megan believed her knowledge of "and" was not relevant in the context of the computer.

Episode 3

Summary of Episode 3

Without support from me, Megan created the representation shown in Figure 30 and then concluded that "true and true" would report true. The transcript during her creation of this representation is not the focus of the analysis, but is provided below.

> *"But if like you have um like I was saying like ok 'A' set 'A' to be, set 'A' to be true if, ok then if block would be above it, if um. Like if wearing red shirt then set 'A' to be true? And then if wearing red shoes, then set 'B' to be true and then it would be else false for both of them."*

Functionally the transcript above narrates the creation of the representation shown in Figure 30. Megan ended this narration with the statement *"and then it would be else false for both of them,"* which does not describe text in the representation. I interpret this statement as indicating that both "if" cases would have an "else" case to set "A" and "B," respectively, to false.

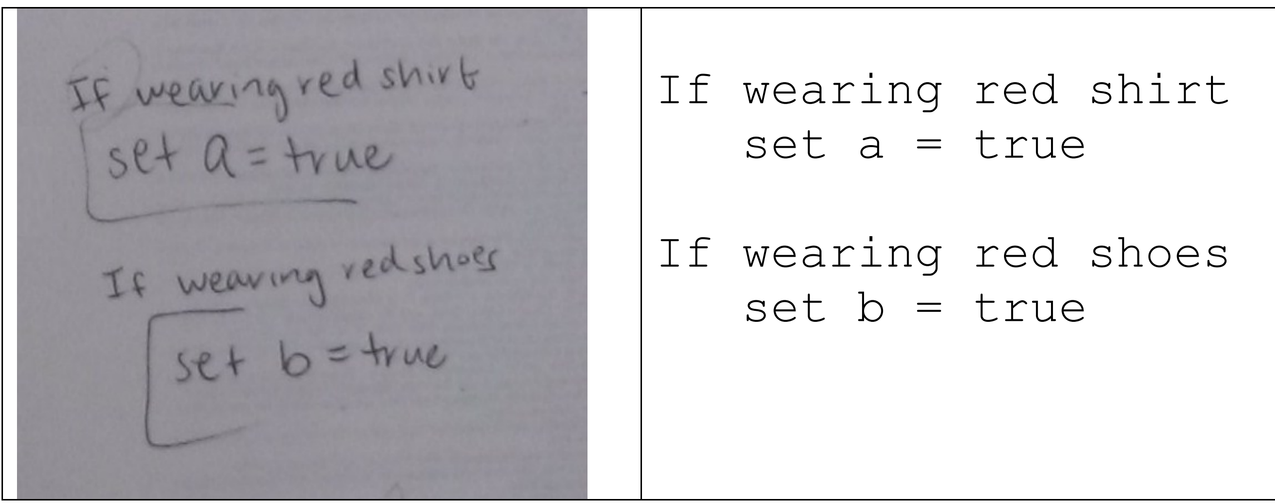


Figure 30. A secondary context created by Megan to consider the case where "A" and "B" are true.

Immediately after she constructed the representation shown in Figure 30 she drew the following conclusions: [2]

> *"But um so then you could have 'A' and 'B.'* ***Oh yeah. Yeah.*** *So then ['A and B'] could be true and so then if wearing a red shirt then 'A' is true and then if wearing red shoes then*

[2] In the following quotations, bold denotes verbal emphasis.

www.manaraa.com

'B' is true. So that would mean that if 'A' is true and 'B' is true, then this would be true // ***Oh so you can have true and true.****"*

I probed for further explanation by asking *"Ok why does that mean you can have true and true?"* The transcript below shows Megan's response to this question, which focused on how "and" includes an implicit "if" component.

"Because um you can like have whatever variables so if you're setting it so basically if this, so this block is saying if this is true and this is true (points to Figure 30), then report the whole thing as true. (Interviewer: Oh ok) But then if this is true and this is false (points to "A" and "B" in Figure 30), then that means both of them aren't like true so then the whole thing would be false."

Next, Megan responded to my question: *"How is what you're saying now different than how you were thinking about it before?"*

"before I was just putting in true and false like setting them to be true and setting them to be false. So like okay true and true make what? But now I realize if you do: ***If*** *this is true and this is true, like that's what it means. Not just like true and true, but* ***if*** *it's true and* ***if*** *it's false. So I think just started thinking of the more if"*

Megan's response focused on how she *"started thinking of the more if"* rather than just "and." During the interview it was unclear to me if Megan realized that the expression we were considering did not include an "if." I mentioned that the "and" expression Megan had written was not accompanied by an "if." I said: *"just this block by itself (referring to the block "A and B") doesn't have an 'if', does that make it not work?"* While Megan's first response was a dejected sounding *"Oh,"* she responded *"Um actually I think it could still work… it's still saying like if 'A' is true and 'B' is false. There's no 'if' though, but I think it works."*

Megan continued by saying *"I think actually in our class they should have explained more how these blocks like work and what the input and output is because I still am confused about it. Like when we use it and right now. (Interviewer: Yeah) But I think it's starting to make sense so you can have it be true and false."*

Analysis of Episode 3

At the beginning of episode 3 Megan wrote pseudocode for a real-world use of conditionals in which the predicate tests the state of an individual's clothing and conditionally sets the variables "A" and "B." From this context, Megan was then able to reason that "A and B" could return true and she again generated appropriate coordination of program state for the case of two true values. Her causal net element that *this world and the computer have different rules* dominated her reasoning in episode 2. In virtue of the bridge she created between this world and the computer, this causal net element is given less priority in her reasoning in episode 3. Unlike episode 2, in episode 3 Megan generated inferential chains to identify the behavior of "and" for various cases. This analysis develops a hypothesis regarding the way in which Megan connected her everyday knowledge and computer science knowledge and how

www.manaraa.com

her conclusions were supported by bridging her structural and functional knowledge of “and” by focusing on “if.”

In episodes 1 and 3 Megan used non-programming specific knowledge to make inferences about state. It is likely that in episode 1 she was unconscious of this out-of-domain knowledge use, but in episode 3 was able to explicitly connect her programming and everyday linguistic knowledge of “and.” She may have developed the causal net element that *this world and the computer work the same way for “and”* by observing that it was possible to represent her everyday example about clothing in Scratch. In episode 2, Megan considered the decontextualized Scratch expressions in Figure 29 without being able to realize the relevance of her everyday knowledge. In episode 3, she demonstrated alignment of state when she connected the decontextualized Scratch expressions to a representation of her real world scenario in Scratch. This may have served to connect her everyday and computer science knowledge or, perhaps more importantly, it made the idea that *this world and the computer work the same way for “and”* more plausible. This connection or this plausibility functioned to enable Megan to use her correct intuition about “and” to predict the behavior of “and” in a computer science context.

I believe that through representing in Scratch psuedocode her real-world example of “and,” which was about clothing, she connected her understanding of “and” from *“this world”* with *“the computer.”* However, the mechanism of this connection and the specific knowledge that Megan bridged from “this world” to “the computer” is not obvious. To explore these issues, I examine the role played by Megan’s everyday and computer science knowledge of “if.”

In Megan’s explanations of her insight, she focused heavily on the role of the “if” in her reasoning about the expression “A and B.” Each of the times that Megan used “and” to describe her real-world example she also used “if.” Megan appears to not only use her everyday knowledge of “and,” but also her everyday knowledge of “if.” Megan re-explained “and” by using “if” to separate possibilities for what “and” would “report.”

> *“This block is saying if this is true and this is true, then report the whole thing as true… But then if this is true and this is false then that means both of them aren’t like true so then the whole thing would be false.”*

The “if” she used here is not the traditional conditional of “if” used in Scratch. This Scratch “if” takes only a single value and based upon that value determines the next line of code to be executed. Contrary to Megan’s description above, the Scratch “if” is not directly involved in the reporting of any value. The Scratch “if” is fundamentally different than the role “if” plays in Megan’s description of “and.” Megan says that the expression “true and true” actually “means” “[i]f this is true and this is true… Not just like true and true, but if it’s true and if it’s false.” In this quotation she contrasts the expression “true and true,” which was stated without an “if,” with a description that uses the “if” to include the idea that decision is made within or by the “and.” Megan’s description of “and” appears to contain an “if” and I will refer to this causal net element *“and” contains an “if.”*

www.manaraa.com

It is possible that this element that *"and" contains an "if"* helps bridge her functional and structural knowledge of "and." Students learn particular ways of accomplishing tasks; for example Megan might have developed functional knowledge for using "and" within an "if" expression. A common structure for an "and" expression would be "if (x = = y and x = = z)." This functional knowledge to create these types of expressions could serve Megan when reasoning about the behavior of "and" within an "if," but might not be applicable to reasoning about the behavior of "and" if the "if" is not present or if the two elements in the "and" expression are Boolean values rather than tests. In the methods section I discussed the common incorrect structural assumption that "if" operates on an expression rather than the Boolean output of an expression. If the conditional expression used in the "if" includes an "and" a student may assume that the "if" is responsible for executing the two tests or possibly responsible for executing the "and" expression. Megan's explanation can be seen as patching this structural model by embedding the assumed responsibility of "if" as an implicit "if" in the "and." My hypothesis is that she sees "and" as requiring a test and by attributing testing to "if" and thinking about "and" as containing an implicit "if" can satisfy her requirement of "and" as containing a test.

Here she seems to be reasoning about the "and" block as if it contained an implicit "if" block. I assume that this causal net element built upon Megan's knowledge of "if," from both inside and outside of the computer science context. During the interview it was not clear to me whether Megan understood that the "and" could exist outside of an "if." Megan eventually acknowledged that the "if" is not necessary, but appears to continue to reason based upon the causal net element that *"and" contains an "if."* She said that *"it's still saying like if 'A' is true and 'B' is false."* I claim that she builds upon some everyday knowledge of "if," but unfortunately we do not have more resolution regarding the nature of Megan's everyday knowledge of "if."

Episode 4

Summary of Episode 4

Megan next continued with a previous line of discussion regarding whether or not the variable "A" could be a number. She concluded *"I don't think it makes sense if 'A' is a number"* and set out to create an example to show that "A" cannot be a number. She constructed the expression "5 and true" shown in Figure 32[3].

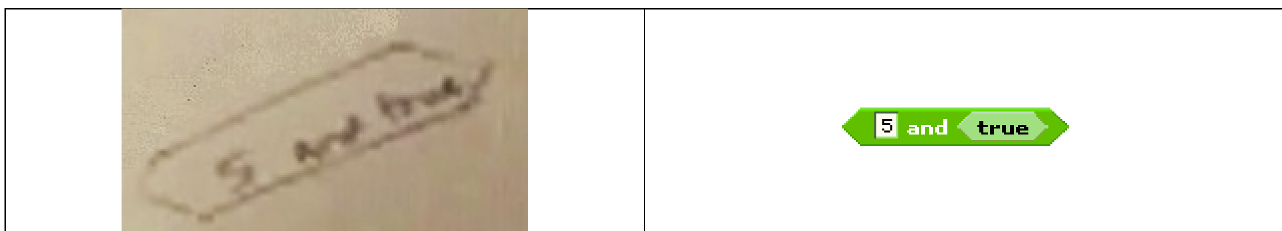


Figure 31. A context created by Megan to consider whether the variable "A" could be a number.

[3] This can be constructed in Snap, a variant of Scratch, but is a syntactically invalid expression because providing 5 as an argument to the "and" function would create an error.

www.manaraa.com

Although she set out to show that it was invalid for “A” to be a number, as Megan talked through this expression she decided that if you had “earlier” set the value of “A” to be 5 that this would return true. Megan said *“you would have like 5 and true? Oh that does make sense, if you set ‘A’ to be 5 earlier, or so like you don’t know if it is but like if ‘A’ is 5 and ‘B’ is true, then this whole thing is true.”*

I had significant difficulty in understanding Megan’s reasoning and after a few inconclusive questions and answers, I asked *“why is ‘5 and true’ not false?”* Megan responded as shown below and added to Figure 32 to create the diagrams in Figure 32 and Figure 33.

> *“it’s only if you have like set ‘A’ equals 5 (generates Figure 32). Or like you have an index and ‘A’ becomes 5. If ‘A’ was like 4 (generates Figure 33) then this part (points to the ‘5’) would be false and this part (points to the ‘true’) would be true and then the whole thing would be false.”*

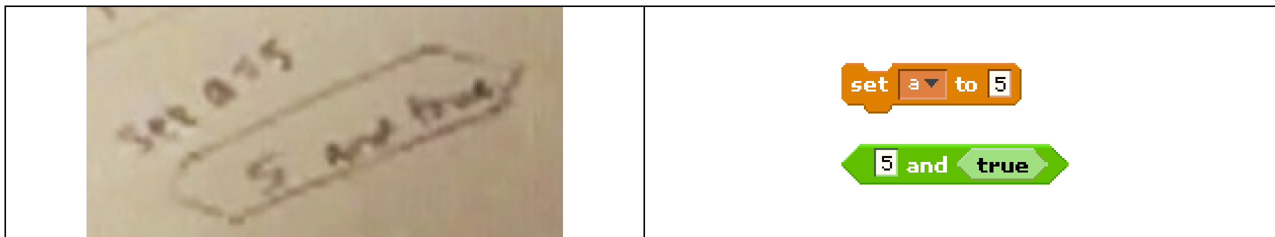


Figure 32. Modified version of Figure 31 to include an assignment to the variable “A.”

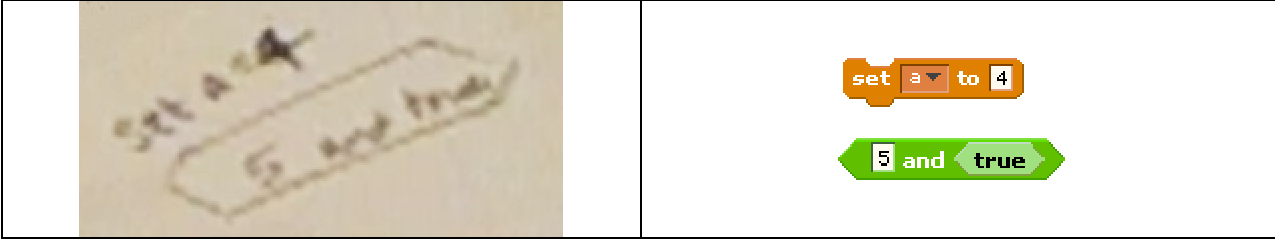


Figure 33. Modified version of Figure 32 that sets the variable “A” to 4 instead of 5.

I asked Megan to explain *“why if ‘A’ would be 4, then it would be false?”* Megan at this point recognized her error as shown in the following transcript. *“Because if um, because this is saying if ‘A’ is 5, oh wait it’s not saying if ‘A’ is 5, it’s saying just 5 … so if oh wait. No that doesn’t work.”* Upon closer investigation Megan realized that *“it’s not saying if ‘A’ is 5”* and she appeared to correctly interpret this expression as invalid. The ellipses in the quotation above indicate that words were removed from the quotation. These words, shown in the following quotation seem to include only fragments of ideas, that are difficult to follow, but clearly show that Megan’s transition from the conclusion from *“it’s saying just 5”* to *“No that doesn’t work”* was not immediate. The phrases that she uttered at this time are as follows, *“Oh could you have, oh well actually in that it has just ‘A’ in here. So if you have ‘A’ in here,”*

www.manaraa.com

Analysis of Episode 4

Megan's statements implied that "5 and true" tests the value of the variable "A," but no test exists in this expression to check the value of the variable "A." The primary question in this analysis is why did Megan originally see the expression "5 and true" as testing whether the value of the variable "A" was 5.

From Megan's performance on other problems I have no reason to believe that she saw the symbols she had written in any other way than they appear; I believe she competently extracts the text "5 and true." However, to interpret that extraction she used elements from her causal net, which explains how her interpretation could differ from an expert's interpretation. In particular, I believe her reasoning was influenced by her causal net element that *"and" contains an "if"* and her strong connection between "and" and "=."

In episode 2, Megan had difficulty seeing the "and" as reporting a value, but in episode 3 determined that it could. The relevant observation from episode 3 is that Megan emphasized the way in which the "and" included the testing aspect of the "if." Megan may have aspects of a structural model that "and" involves testing. In episode 3, she saw "and" as including an implicit "if" and similarly in episode 4 she may see "and" as including an implicit "=." This process of attributing implicit behavior to "and" is consistent with incomplete structural knowledge. A correct structural model of state would imply that the "and" does not have access to test the value of the variable "A" and an implicit test does not take place.

Megan had a strong connection between "and" and "=," which I believe influenced her reasoning in episode 4. At one point earlier in the interview Megan flipped to talking about the "and" as "=." This flip happened when she discussed what "and" would report if provided two false values, which is the only case that differs between "=" and "and." She quickly caught herself, but momentarily entertained the idea of the "and" being, or being replaced by, an equals function. She says *"I think it would report false, but then if this equals then it would report true, but it's not equals."* This strong connection or even perhaps reliance on "=" may have occurred throughout the interview.

The causal net entity of *"and" means both of them* does not specify that "both of them" need to be true. Another interpretation is that "both of them" must be the same. This interpretation, which is equivalent to "=," correctly predicts the behavior of "and" for all cases but the case of two false values. "False and false" reports false, but "false = false" reports true.

I conclude that Megan sometimes used the resource I will refer to as *"and" tests*. This aligns with thinking about an "and" as like an "=." It is possible that Megan used this causal net element when she concluded that "5 and true" reports true and that "4 and true" reports false. Although Megan later recanted these claims, she appears to be guided by the belief that an "and" tests something about the current state, which can be partially explained by a strong connection to "=."

www.manaraa.com

Discussion

In the focal episodes, Megan was reasoning about the behavior of “if” and “and,” but from a computer science perspective the relevant “behavior” that she considered was the possible state of the variables “A” and “B,” the input and output states of the conditional “and,” and the input and output states of “if.” The analyses of these episodes described the interaction of elements of Megan’s causal net, which may help understand the role of everyday knowledge in reasoning about computer program state.

In episode 1 Megan was able to articulate the behavior of the “and” function for all possible initial states. While Megan did not appear confident in these facts, she demonstrated appropriate coordination and appeared to have an understanding of the expression “A and B.” In this episode she appeared to generate appropriate coordination through the causal net elements of *“and” means both of them* and *“and” works to combine*, as well as knowledge about the relevant sets of inputs to “and.” Episode 1 shows a real-world example of “and,” and other causal net elements of “and,” which are rooted in Megan’s everyday knowledge.

In episode 2 she rejected that the same expression “A and B” would *“report anything.”* This conclusion prevented her from coordinating state and she appeared to use a causal net element of *this world and the computer have different rules.* Episode 2 shows an example of a lack of span and shows Megan’s sensitivity to perceived context as relating to “this world” or “the computer.”

In episode 3 she created in Scratch a representation of an everyday example about clothing. From this bridging of her computer science and everyday knowledge, Megan was again able to demonstrate appropriate coordination and extended the span of her coordination class of state. In addition to the causal net elements identified in episode 1, Megan appeared to build upon the causal net elements of *this world and the computer work the same way for “and”* and *“and” contains an “if.”* This is an example of a student using intuitive knowledge productively to reason about a computer science context. However, throughout this progression Megan did not appear to recognize an important property about “if” and “and,” which is that they both operate only on Boolean values and not expressions.

In episode 4 Megan temporarily believed the expression “5 and true” to be testing the value of the variable “A.” Eventually Megan caught her mistake, but before this she appeared to be reasoning based upon the causal net elements of *“and” contains an “if”* and *“and” tests*, as well as a strong connection between her knowledge of “and” and “=.”

My analysis extends coordination class theory to the domain of computer science for the first time by using coordination class theory to analyze a student’s reasoning about the concept of state in a computer science context. This moves computer science education forward by providing an initial theory describing the moment-by-moment interaction of knowledge. This work advances coordination class theory by evaluating the theory outside of physics and mathematics

www.manaraa.com

Relationship to Previous Coordination Class Research

In the following section I will draw out some similarities and differences between this work and previous coordination class analyses.

diSessa and Sherin (1998)

The article by diSessa and Sherin (1998) introduced the term coordination class and diSessa and Sherin (1998) used examples from an undergraduate student taking an introductory physics course. The third empirical example in diSessa & Sherin (1998) focused on a student's coordination class of acceleration and coordination class of force. She correctly identified that a book pushed by the interviewer across the table was not accelerating. In this way she correctly coordinated acceleration. However, she demonstrated incorrect coordination of forces. Instead of using the equation relating acceleration and forces from her causal net, F=ma, she uses intuitive knowledge that generates a non-normative result in this context. The intuitive knowledge she used was that "*contact conveys motion*," which diSessa (1993) identified as a p-prim. This intuitive knowledge and her confidence in the lack of acceleration of the book leads her to conclude that the equation F=ma does not apply in this context. She said *"I guess you can just say that, you know, those darn equations aren't applicable to every single thing. They're not always true. You can't live by them… I just thought that F=ma was one of those that was universal."* This student's rejection of the relevance of the equation F=ma to this situation can be seen as parallel to Megan's rejecting that "true and true" would not report anything. The student in diSessa & Sherin (1998) used the idea that "contact conveys motion" instead of the equation "F=ma." Megan rejected that "true and true" would report anything based upon the idea that *this world and the computer have different rules* instead of her knowledge that *"'and' means both of them"* and that *"and" combines*. Both students cued relevant knowledge that could have been productively applied, but reject that this knowledge is relevant in this context. The differences between these two cases are numerous, but these few commonalities are remarkable to see across domains.

diSessa and Sherin (1998) identify the ways in which their participant was justified in being careful about the applicability of equations. diSessa and Sherin note that F=ma does in fact have limitations in terms of applicability and her sensitivity that it may not be universally applicable, although it does apply to the context she discusses and is part of a generally productive attitude for reasoning about physics. Similarly, Megan rejects that her everyday knowledge of "and" is relevant to the computer science context. She too identifies that there may be different contexts of applicability and questions why "and" in the real world and computer would behave in the same way.

Parnafes (2007)

This analysis in this chapter diverges from previous coordination class analysis by focusing on Megan's coordination class of state independent of what type of information believed she was determining. This divergence can be seen most clearly through a comparison to the analysis of Parnafes (2007). Parnafes (2007) discussed students' understanding of the intuitive coordination class of fastness. She discussed how students originally used this coordination class to analyze oscillations when in fact they should be using the coordination

www.manaraa.com

classes of frequency and velocity. A heuristic of selecting an appropriate coordination class that has guided previous coordination class analyses (A. A. diSessa, personal communication, April 3, 2012) is that the relevant coordination class is the coordination class that the participant believes themselves to be using. This governed the development of the construct of intuitive coordination classes (Parnafes, 2007; A. A. diSessa, personal communication, April 3, 2012) where students' believed that they were identifying the relative "fastness" of the oscillator. I believe that this artificially emphasizes a transition from the intuitive coordination class to the correct coordination classes of velocity and frequency. Many of the knowledge elements that the participants used across the episodes presented were likely the same or quite similar. I do not claim that Megan was aware that she was identifying state during these episodes. I deemphasize what coordination class the individual believes themselves to be using for the purpose of emphasizing the commonality between the tasks and the ways in which Megan bridges her everyday knowledge to reason about computer science.

Levrini & diSessa (2008)

Like the students in Levrini & diSessa (2008), Megan demonstrated in episode 2 that students can reason about the relevance of various knowledge elements regarding state; however, I am not implying that this process of articulate alignment will be common or necessarily viable pedagogically. While other conceptual change research (*e.g.*, Strike & Posner, 1992) prescribes pedagogical techniques to confront inconsistency in students' understanding, I do not offer sufficient evidence to suggest that this would be a generally useful strategy. I expect that the majority of students engage unconsciously in their application of relevant knowledge, which might make the technique ineffective. As a second point, this form of confrontation may engender in students a lack of trust for their ideas (Smith, diSessa, & Roschelle, 1993), many of which could serve as fertile grounds for the development of coordination.

Pedagogical Implications

Although the primary focus of this paper is the development and refinement of theory regarding computer programming knowledge, I highlight some of the pedagogical implications from my work.

A first pedagogical implication of this works is to join other researchers (diSessa, 1986; du Boulay, O'Shea, & Monk, 1989; du Boulay, 1989; Cooper, Dann, & Pausch, 2000; Ben-Ari, 2001; Sajaniemi & Kuittinen, 2005; Shinners-Kennedy, 2008) who focus attention on state as an important concept in computer science education. The case presented in this chapter demonstrates the complexity of this concept in the computer science context and showed interactions between a student's structural and functional knowledge (diSessa, 1986) and everyday and computer-science-specific knowledge. This complexity suggests that a pedagogical focus on state might be necessary or at least productive for students. This analysis provides an empirical basis for the conclusions of Shinners-Kennedy (2008) that individuals have experience with state that may be relevant to computer science contexts. In episode 3 Megan achieved the type of transfer of her everyday knowledge to the computer science context that we hope to achieve in instructional contexts.

www.manaraa.com

A second pedagogical implication of this work is that coordination of knowledge at one point in time does not guarantee coordination of the same knowledge at another point in time. Coordination class theory argues that knowledge within a coordination class is made up of a variety of conceptual elements. With improved coordination students access these elements more reliably to determine the relevant focal information. A key component of the theory is that errors in performance are not necessarily evidence of a lack of knowledge. Frequently the individual could be described as "having" relevant knowledge that they do not use within a context. In the dynamic process of problem solving, students make conscious and unconscious decisions about what knowledge they apply to a problem. A novice may have the relevant knowledge, but may not, for some reason, use the necessary knowledge in concert to correctly identify the focal information of the coordination class. The case presented here shows examples where Megan originally used intuitive knowledge, but had difficulty applying that knowledge in a second context. She was eventually able to create a bridge to be able to use this knowledge in both cases.

A third pedagogical implication of this work is that tracking program state requires an extensive set of facts regarding the programming language as well as expertise in utilizing these facts in concert. The case study showed that the central challenge was the coordination of everyday and computer science knowledge and not, as others might expect, simply the acquisition of knowledge regarding the programming language. I hypothesize that errors where students have the requisite facts necessary to have prevented or detected their errors are frequently interpreted by students and instructors as unproblematic mistakes that do not require remediation. For example, Megan's reasoning about the expression "5 and true" as testing the value of the variable "A" can be seen as only a simple mistake. The coordination class analysis provides a focus on the coordination of relevant knowledge and suggests that these incidences are not simply mistakes and are best categorized, more particularly, as evidence of a lack of span or alignment.

www.manaraa.com

PARTIAL DESCRIPTIONS OF STATE CHANGE

The previous chapter tracked the moment-by-moment use of knowledge when an individual used the coordination class of state to reason about the behavior of the conditional "and." In this chapter, I identify a particular type of inference that an individual can make when reasoning about state. The emphasis in this analysis is not on the dynamics of how these inferences are linked to create an inferential chain, but instead on developing a model of a particular type of link that can exist in a concept projection of state.

This chapter is focused on students' statements that summarize patterns of state and state change as a type of inference than an individual can make when reasoning about state. Researchers have reported that students have difficulty providing a summary of code that focuses on the overall behavior of the code rather than the line-by-line details (Hoadley, Linn, Mann, & Clancy, 1996; Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009). Researchers have speculated that the ability to produce a summary of code develops after the ability to trace code (Venables, Tan, & Lister, 2009).

Despite the reported difficulty, I found that students frequently made statements that summarized aspects of state when answering a question about how to avoid an infinite loop in a recursive function. Surprisingly, many of these students had difficulty on a previous problem that required tracing the same recursive function. This pattern of students' competence provides the opportunity to investigate a context in which students were successful at generating a summary of code that does not focus on the line-by-line details and the opportunity to schematize this type of inference and how it relates to the process of tracking program state. The generalization that students have difficulty providing a summary of code requires additional refinement to provide contextual specification.

The data from this study were taken from seventeen clinical interviews with college students who were enrolled in an introductory programming course at the University of California, Berkeley. During the approximately hour-long interviews, participants talked aloud while they solved computer programming problems.

The problems used in the interviews were translated versions of the problems identified by Reges (2008) as the five questions that were most highly correlated with success on the 1988 Advanced Placement Computer Science (APCS) exam. The selection of these questions aligned the study's interview with content from an international introductory computer programming curriculum. Another benefit of selecting questions most highly correlated with success on the exam is that these questions may contain a set of competencies that are important for introductory programming.

In the following analysis, I focus on one such question, which asked students to identify the conditions that would *not* create an infinite loop in a recursive function. This question, which I call the "infinite-loop" question, followed a question that asked students to find the

www.manaraa.com

output of the same recursive function for a particular input, which I call the "tracing question." The tracing question and the infinite-loop question are shown in Figure 34 and Figure 35 and are described in detail in the methods section of this dissertation. The tracing question was not one of the questions most highly correlated with success on the exam, but was included in the interview because it was the first part of a two-part question, where the second part was one of the questions most highly correlated with success. During the interview, participants solved these problems in the same order as they appeared on the APCS exam and participants' responses to this pair of questions form the data corpus of the study.

```
(define (whatIsIt x n)
   (if (= n 1)
        x
       (* x (whatIsIt x (- n 1))) ))
```

What value is returned by `(whatIsIt 4 4)`?

A) 8 B) 16 C) 24 D) 64 E) 256

Figure 34. The "tracing question": a translation of a question from the 1988 APCS exam.

Which of the following is a necessary and sufficient condition for the function `WhatIsIt` to return a value if it is assumed that the values of n and x are small in magnitude and are both whole numbers?

A) $n > 0$
B) $n = 0$
C) $n > 0$ and $x > 0$
D) $x \leq n$ and $n > 0$
E) $n \leq x$ and $n > 0$

Figure 35. The "infinite-loop question": a replication of a question from the 1988 APCS exam.

On the infinite-loop question, participants demonstrated accurate reasoning and made statements about the patterns of state change. This contrasted with many participants' performance on the tracing question, where they were not successful tracking state.

To introduce the nature of participants' insights on the infinite-loop question, I provide a case study of one participant's solution to the two focal questions. This case study of the student Rick[4] (participant identifier: Yellow_BL/Purple_BR) shows that the context of the infinite-loop question induced a specific insight that appeared to be missing in his solution to the tracing question and this missing insight was likely the reason for his difficulty. I selected this case because it clearly demonstrated difficulty on the tracing question and insights about state change on the infinite-loop question.

The primary contributions of this chapter are operationalizing this type of state summary, which I call a "partial description of state change" and demonstrating the existence

[4] All names are pseudonyms

المنارة للاستشارات
www.manaraa.com

of this competence within a particular context. The case study provides two examples of a partial description of state change. After orienting the reader to the type of inferences observed in the data, I delineate and describe two types of partial descriptions of state change. This is followed by a collection of quotations from across the data corpus, which serves primarily as exemplars to help illustrate the range of statements that would be classified as partial descriptions of state change.

A later chapter includes a discussion of some pedagogical implications for teaching recursion. There I discuss the possibilities of encouraging students to reason about the cases in which a function call results in an infinite loop before attempting to trace the same function.

Case Study

When creating content logs for the data, I noticed that the participants occasionally had greater insight regarding the behavior of the `WhatIsIt` function when answering the infinite-loop question than when answering the tracing question. The following case study demonstrates this pattern. This case is used to make the claim that the insights on the infinite-loop question are relevant to the coordination of state before introducing a classification system for these insights. The analysis of this case includes a narration and interpretation of the participant's solution to both the tracing question and the infinite-loop question. The data from this participant is then used to map the ways in which his insights on the infinite-loop question might have been helpful to him when solving the tracing question.

For the participant Rick, the infinite-loop question appeared to elicit insights about the patterns of state change that were not accessible to him when he solved the tracing problem. As a brief overview, he mentioned accurate patterns of state change for the variables x and n, but his insufficient tracking of these same variables was the likely cause of his incorrect solution to the tracing problem. The section following the case study seeks to schematize more generally the nature of his insights.

Rick began the tracing question by explaining that he had recently learned recursion and then read the question aloud. After a brief pause, I asked Rick *"What are you thinking?"* He responded with the statements below and generated the first line of handwritten inscriptions in Figure 36, "4 3 2 1."

> *"Oh okay – so you're multiplying x, which is the first number here (writes 4) by um, 3 (writes 3) and then 2 (writes 2) and then 1 (writes 1)."*

```
(define (whatIsIt x n)
   (if (= n 1)
         x
        (* x (whatIsIt x (- n 1)))
))
```

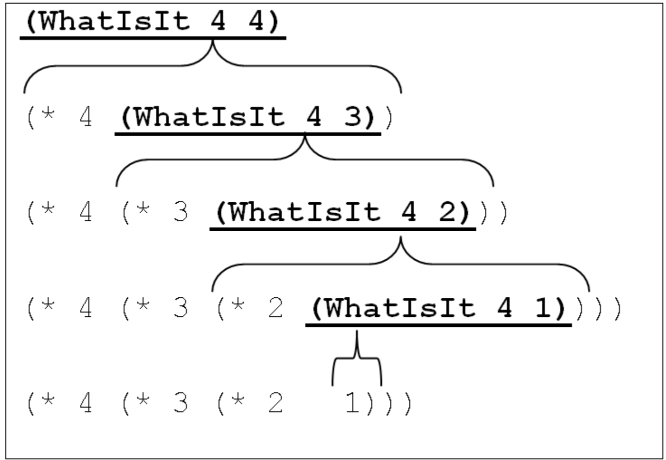
Figure 36. Notes made by Yellow_BL when solving the tracing problem

www.manaraa.com

Without speaking, Rick generated the second line of handwritten inscriptions in Figure 36. Although I do not have evidence of his reasoning, he wrote 12, the partial product of 4 and 3, and 2, the partial product of 2 and 1. His final answer was 24 (or 4x3x2x1), instead of the correct answer, 256 (or 4x4x4x4).

The variable that gets multiplied by the function is, as Rick said, x. However, x is always 4. Rick's reasoning was consistent with a function that continually multiplies the variable n, because the value of n begins with a value of 4 and decreased by 1 in each recursive call. He may have incorrectly believed that the value of n gets multiplied by the function or he may have incorrectly believed that the value of x was changing. Given his statement "so you're multiplying x," the latter seems more plausible, but ultimately the source of his mistake is unknown. Instead it might be that when tracing the function he was not paying close enough attention to how the variables change and how the variables are used. When he traced through the function he did not mention the individual function calls, which may be a symptom of his lack of care in tracing the values of the variables x and n and may have prevented his coordination of state. A formal coordination class analysis is not conducted here, but it may be helpful for the reader to note that here Rick demonstrated a lack of alignment in his coordination class of state. Rick applied his partial coordination class of state to determine the return value of the function, but his determination of the state was inaccurate. This alone classifies as a lack of alignment, but in his concept projection of state his partial descriptions of state were not consistent with his written representations, which is further evidence of a lack of alignment.

Figure 37 shows one set of calculations that can be inferred from Rick's statements. Given the fact that he answers the question incorrectly, it is ambiguous whether he incorrectly tracked the values provided to the function or incorrectly tracked how those values were used in the function. This diagram includes the assumption that Rick correctly tracked the values of the variables provided to each recursive call. While he identifies himself as having *"just learned"* recursion, I infer from his solution that he correctly combines pending calculations from each level of the recursive calls, which has been identified as a difficult aspect of tracing recursive functions (*e.g.*, Kahney, 1989).



www.manaraa.com

Figure 37. Incorrect recursive tracing that was inferred from statements made by Rick (YellowBL/PurpleBR) when tracing a call to (WhatIsIt 4 4)

I expect the reader might question the severity of the mistake made by Rick. He may have just confused the values of the variables x and n. This may be a completely accurate description, but is at best an incomplete explanation. When solving other problems he did not appear to confuse the values of any variables. It appears from the data that this confusion is context sensitive. This simple confusion may be evidence of a more systematic weakness in tracing recursive functions. This relates to the larger theme in the work of the coordination class of state. An expert's coordination class of state includes both the ability to correctly track state and the knowledge that precise tracking of variables, such as with a representation, is necessary to correctly track state.

Next, on the infinite-loop question, Rick's reasoning included statements about the pattern of state change between recursive function calls. After reading the question aloud to himself, Rick quickly determined the correct answer. He said:

> "*So here, uh the base case is n equals one and obviously you're subtract, you end up subtracting here, so it can't be less. It can't have anything where n is less than one. So n has to be greater than zero, so that gets rid of that (crosses off answer option B) and, (pause) so both are whole numbers so that means that um x does not it doesn't matter what x is here, so n has to be greater than zero (circles A).*"

Rick went on to elaborate on his answer. Most notably he stated that *"you end up just multiplying whatever x is by this recursive call"* and also that *"you're not changing x."*

Rick's statements about the patterns of state change between recursive function calls contradicted his calculation on the tracing question. In that problem, it appeared that he assumed either that n was being multiplied or that x was changing. Both of these hypotheses seem refuted by his statements on the infinite-loop question. However, we can form a different interpretation of this seemingly inconsistent behavior by taking a perspective from coordination class theory. With this perspective we can map the two questions to different contexts that in turn elicited different knowledge.

Rick did not notice the contradiction of his statements on the infinite-loop question and his solution to the tracing problem. However, his insight could have served as a check to his line-by-line tracing of the code. Rick's knowledge that supported the inference that the value of the variable x is not changing could have been fruitfully applied when tracing through the same function. It may be that competent individuals make some mistakes like Rick's, but that they are more deliberate about checking their solution and more capable of checking their solution with another method.

From the Knowledge in Pieces perspective, it is not surprising that students built upon different knowledge on two similar problems. The tracing question appeared to focus Rick on specific values of x and n and the infinite-loop question appeared to focus Rick on these more general patterns of state change.

www.manaraa.com

It was not relevant to describe how the variable x changes to justify his solution to the infinite-loop question, but his descriptions of how x changes seemed to flow naturally within the interview. The think-aloud format can make tangential inferences like these explicit when an individual says them aloud. There may be different supports for this type of tangential, but productive, inferences within a think-aloud and within a silent assessment.

Case Study Conclusions

This case showed an example where a participant stated insights regarding the patterns of state change. These insights mapped directly to the difficulties the participant appeared to have on the tracing question. He was unsuccessful tracking the state of the variables x and n, but then had correct insights about how the state of these variables changed. The content of his insights on the infinite-loop question maps directly to his weakness on the tracing question.

I am not making the argument that the participant's insights on the infinite-loop question would be the only path to his success on the tracing question, only that there is a mapping between this participant's insights on the infinite-loop question and his apparent weaknesses on the tracing question. None of the participants returned to the tracing question after answering the infinite-loop question; therefore it is not possible to demonstrate that Rick's or other participants' insights were in fact productive, only that they might have been productive. The ability to use multiple methods to check an answer may be a type of expertise and these insights would be productive for that purpose.

I also do not claim that the participant would necessarily have had the same insights on the infinite-loop question if he had not first attempted the tracing question. It is possible that the participants developed a greater understanding of the `WhatIsIt` function by attempting to trace it.

Previous Research

There is great interest in understanding students' difficulty explaining code beyond line-by-line descriptions (Hoadley, Linn, Mann, & Clancy, 1996; Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009). Much of the research in this area has been conducted by research from the BRACElet project, which has investigated the hypothesis that there exists a hierarchy of programming skills (Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009). Lopez, Whalley, Robbins, & Lister (2008) found a positive correlation between participants' performance summarizing code and their performance writing code, which was statistically significant at the .01 level. They used these data as partial evidence for the existence of a hierarchy of programming skills. Venables, Tan, and Lister (2009) describe this hierarchy of programming skills in the following quote:

> "First, the novice acquires the ability to trace code. As the capacity to trace becomes reliable, the ability to explain code develops. When students are reasonably capable of both tracing and explaining, the ability to systematically write code emerges." (p. 128, Venables, Tan, & Lister, 2009)

www.manaraa.com

Whalley *et al.* (2006) attempted to categorize the level of abstraction in participants' attempts to "Explain in plain English" what a particular segment of code "does." They used the Structure of Observed Learning Outcome (SOLO) taxonomy (Biggs 1982, Biggs 1999) to perform this categorization. The SOLO taxonomy was designed to describe stages in students' learning within a domain. The bullets in Figure 38 show the five levels of understanding identified by Biggs (1999) that may serve as a reminder for those familiar with the SOLO taxonomy. These descriptions may be insufficient for a reader unfamiliar with the SOLO taxonomy, but only two categories are prominently featured in previous computer science education research and are elaborated in the following paragraph.

- Pre-structural - The task is not attacked appropriately; the student hasn't really understood the point and uses too simple a way of going about it.
- Uni-structural - The participant's response only focuses on one relevant aspect.
- Multi-structural - The participant's response focuses on several relevant aspects but they are treated independently and additively. Assessment of this level is primarily quantitative.
- Relational - The different aspects have become integrated into a coherent whole. This level is what is normally meant by an adequate understanding of some topic.
- Extended abstract - The previous integrated whole may be conceptualized at a higher level of abstraction and generalized to a new topic or area.

Figure 38. The five levels of understanding from the SOLO taxonomy, quotations from Biggs (1999)

To make the SOLO taxonomy accessible to computer science educators, Whalley *et al.* (2006) provided descriptions of the categories as they relate to types of questions from computer science. For example, they operationalized the SOLO taxonomy category of *Relational* as "Provides a summary of what the code does in terms of the code's purpose" (p. 248, Whalley *et al.*, 2006). The Relational category is emphasized in their work as a target for participants' explanations. The other SOLO category emphasized in their work is the Multistructural category, which they describe as when a "line by line description is provided of all the code. Summarization of individual statements may be included." (p. 248, Whalley, *et al.*, 2006).

In this line of work, which investigates students' ability to provide summaries of code, researchers frequently used the SOLO taxonomy to rate the quality of participants' code summaries (Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009). These descriptions seem to assume that a participant's responses would fall into a single category and that a response would include either a line-by-line description of the code or would include a summary of the code's purpose. If a student's response included elements matching the Relational category and other elements matching the Multistructural category, it would be difficult to classify the response with one of these categories. Venables, Tan, and Lister (2009) found that it was difficult to reliably categorize students' responses. An expert would, by definition, be able to provide both forms of explanation; therefore it is easy to imagine that a response could match both the Relational and Multistructural category descriptions.

www.manaraa.com

While Venables, Tan, and Lister (2009) argue that there exists a hierarchy of programming with tracing, describing and writing code, they also acknowledge some limitations of their work. Their data might exhibit the same patterns if the tracing questions happened to be the easiest, followed by the code explaining questions, and with the code writing questions as the most difficult. Venables, Tan, and Lister (2009) acknowledge this threat to validity and also note that the correlations were "particularly sensitive to the specific questions asked." (p. 117, Venables, Tan, & Lister, 2009). This is consistent with the pattern observed in this study where a particular context elicited a greater number of statements that could be classified as Relational.

The work of previous researchers (Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009) and the SOLO taxonomy both highlight what I believe to be an important conceptual challenge for students learning computer programming, which is to articulate a description of the behavior and goals of computer programs. The goal of the following section is to characterize the nature of some of the insights the participants had when answering the infinite-loop question. For this purpose, the descriptions of the categories Relational and Multistructural provided by Whalley *et al.* (2006) are unfortunately too coarse.

Types of Partial Descriptions of State Change

I will identify two types of what I refer to as a "partial description of state change." The first type of partial description is what I refer to as a "single-line summary." This is essentially a description of how a single line of code modifies state, which has been identified as central to programming competence (diSessa, 1986; du Boulay, O'Shea, & Monk, 1989; du Boulay, 1989).

The second type, which is the focus of this work, was inspired by research from the BRACElet project (Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009), but in my analysis I do not use the SOLO taxonomy directly. The BRACElet project research referenced above emphasized students' recognition of patterns of program execution that extend beyond the behavior of a single line of code. I build upon this emphasis. This is not to diminish the importance of understanding the behavior of a line of code both functionally and structurally (diSessa, 1986). However, building upon participants' understanding of a single line of code, it is possible to identify the behavior of multiple lines of code rather than just a single line of code.

I will refer to statements that describe the cumulative behavior of multiple lines of code as a "multiline summary." When I refer to "multiple lines," I do not intend to imply that those lines are unique. For example, consider the code in Figure 39. The first line could be summarized as "Add 1 to the value of the variable x." This summary describes the behavior of a single line of code. An example of a multiline summary would be if the four lines of code in Figure 39 were described as "add 4 to the value of the variable x." This describes the execution of the same line of code multiple times and is a multiline summary about the cumulative behavior of those lines of code.

www.manaraa.com

```
change x by 1
change x by 1
change x by 1
change x by 1
```

Figure 39. Code that adds 4 to the value of x.

Here there were sequential copies of the same line of code. A single line of code can also be executed multiple times when a line of code appears within a loop or a recursive function and the executions of this line may be only a subset of the code that is executed. Another example of a multiline summary is when an individual describes the behavior of a single line of code that is executed multiple times because it appears within a loop or in the body of a recursive function.

I claim that participants provided multiline summaries and I develop this claim through two subsections, which analyze participants' statements regarding the variables n and x, respectively. Quotations were selected for analysis in each of these subsections if they included reference to the variables n and x, respectively. I provide examples multiline summaries and quotations that are similar, but that I do not classify as multiline summaries.

I expect that multiline summaries can be produced by experts and may be particular relevant for successfully tracing recursive functions. This may be a relevant computer science competence that is difficult to acquire. These partial descriptions are inspired by the work of researchers who apply the SOLO taxonomy to computer science (Whalley *et al.,* 2006; Philpott, Robbins, & Whalley, 2007; Lopez, Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009). While there is not a direct mapping between the SOLO taxonomy categories and my definition of a multiline summary, this previous research provided validation for the importance of what I define as multiline summaries. The analysis of these quotations is followed by an argument regarding the potential applicability of these multiline summaries to coordinating state.

Analysis

Participants' Partial Descriptions of State Change for the Variable n

Research from Whalley *et al.* (2006) suggests that participants' summaries of code that do not focus on a line-by-line description of the code are rare. The quotations below are perhaps distinctive as they focus on the changes in the value of the variable n and not the behavior of a single line. To make my definition of multiline summaries clearer I will describe the ways in which the first four quotations can be classified as examples of a multiline summary and the ways in which the fifth quotation cannot be classified as such. These examples are intended to show the range of multiline summaries. I have selected cases that less directly qualify as multiline summaries to show the boundaries of the classification scheme. Despite the diversity among multiline summaries that I demonstrate here, I argue that these multiline summaries are a coherent and observable artifact. This contrasts with the SOLO taxonomy, which is not a sufficiently precise analytic tool for the goal of the chapter to characterize

www.manaraa.com

participants' insights. In the following analysis I identify the quotations as examples of multiline summaries and/or examples of Relational or Multistructural statements for the purpose of demonstrating the differences between these classification systems.

1. "you repeatedly subtract one from n" (Brown_TR)

2. "we're counting down to one" (Orange_TL)

3. "n only gets smaller as you keep going" (Orange_TR)

4. "it will turn negative" (Purple_TR)

5. "you start with n and then you'd n minus one and then once you do that then subtract one from whatever you get and keep going." (Rick: Yellow_BL)

The first quotation describes the process of *"repeatedly subtracting one from n."* Even though *"subtract 1 from n"* describes a single line of the code, with minimal rephrasing from the actual syntax, the use of the word *"repeatedly"* refers to the combined action across multiple function calls and therefore is an example of a multiline summary. This quotation falls short of the SOLO taxonomy category of *Relational*, which Whalley *et al.* (2006) describe as "provides a summary of what the code does in terms of the code's purpose." While the participant summarizes the code, this summary is not strictly "in terms of the code's purpose." Recall that Whalley *et al.* (2006) describe the SOLO taxonomy category of Multistructural as characterized by a "line by line description is provided of all the code." This quotation would not be classified as *Multistructural*, because the participant does not describe all lines of code.

Similarly, the phrase from the second quotation of *"counting down"* refers to multiple executions of a process of counting down and is an example of a multiline summary. Unlike the first quotation that included a minimally rephrased description of a single line of code, this second quotation describes subtracting one as *"counting down."* Coincidentally, describing the process of subtracting one as "counting down" maps reasonably well to the SOLO *Relational* category because, as described by Whalley *et al.* (2006), the participant summarizes the line of code "in terms of the code's purpose," which could be described as to count *"down to one."*

The third quotation does not explicitly mention the operation of decreasing the value of n, only that *"n only gets smaller."* This again provides a summary across multiple recursive calls of how the value of n changes and therefore is a multiline summary. However it does not reference a goal or include summaries of all lines of code and is therefore not classifiable as Relational or Multistructural, respectively. This third quotation is also less specific than the first or second. It indicates the direction of change for the value of the variable n and not the magnitude. In some cases this lack of specificity may be productive for reasoning about parts of code in which the details omitted are not important.

www.manaraa.com

The fourth quotation is that *"it will turn negative,"* which describes the changes in value of the variable n. Unlike the previous quotations, this does not mention a continuing process created by multiple function calls. Instead, this identifies a specific point within the sequence of recursive calls at which point the value of the variable n *"will turn negative."* The point at which this transition occurs can be mapped to a specific line of code. However, this specific point where the value of the variable n becomes negative is within a progression of function calls. This context makes this not only a description of a single line of code, but how that line functions within a larger context and therefore is a multiline summary. Like the third quotation, this quotation is not classifiable as either Relational or Multistructural.

More than the other examples, the final quotation describes a specific line of code. The participant included the phrase, *"and you keep going"* in reference to the process of subtracting one, but the language in this quotation is more directly tied to a single line of code and a single transition to a second recursive call. For example, two executions of the same line of code are mentioned, first that "you'd minus one" and then that you'd "subtract one from whatever you get." I classify this as a sequential set of single-line summaries. This quotation is included to show an example of a summary that includes multiple lines of code, but I do not classify this as a multiline summary.

This set of quotations showed multiline summaries that were and were not possible to classify as Relational and Multistructural. This serves to show some of the inconsistency in the SOLO taxonomy and the need for an additional theoretical term to describe what may be an important inference in tracking program state.

Participants' Partial Descriptions of State Change for the Variable x

It was common for students to make generalizations regarding the value of the variable x on the infinite-loop question. Recall that whether or not the function produces an infinite loop is independent of the value of the variable x. However, the value of the variable x is not irrelevant in the context of tracing the function because the return value of the function is the value of the variable x raised to a power. In the previous section I analyzed the extent to which the participants' statements provided a multiline summary and not a single-line summary or a sequential set of single-line summaries about the state of the variable n. Participants' summaries regarding the variable x did not have the same form. Here I justify why this full set of quotations should be classified as multiline summaries, rather than justifying the classification of individual quotations.

Participants typically described the independence of the value of the variable x and whether or not the function produces an infinite loop, but did not connect that to a specific line of code. Only two of the quotations below provided a reference to a line of code. The quotation listed second to last says *"you're not changing x."* This is an accurate statement because each recursive call passes the unchanged argument x to the function. Although the participant does not mention a specific line of code, no other lines change the value of x and therefore a single line of code is responsible for the truth of this statement. The only other statement that includes even an indirect reference to a single line of code is the participant that said *"so it's n*

www.manaraa.com

that matters," which can be seen as making reference to the base case test (= n 1). The important pattern to note is that few of the participants' summaries of the variable x include references to specific lines of code.

The question becomes whether these statements regarding the variable x should be classified as multiline summaries. Although most participants do not reference the line of code that guarantees that the value of x remains constant, it is the repeated execution of a single line that provides the behavior that is then described by the participant. Therefore I classify these statements as multiline summaries because of the content of each describing the role or behavior of the variable x. The definition of multiline summaries is intentionally quite broad. The higher-level category of partial descriptions of state change includes both multiline and single-line summaries and I expect that both of these sub-categories could be further subdivided.

I outline some of the patterns from among the participant quotations regarding the variable x, presented in the same order as the quotations shown below. The patterns within these quotations help again to demonstrate the range of examples of multiline summaries. One of the expected dimensions of my definition of the multiline summaries construct is that identifying a particular statement as a multiline summary does not indicate that this summary is correct or faithfully explains the behavior of multiple lines of code. A number of the following quotations demonstrate this feature and instances are noted below.

1. "x could be anything" (Brown_TL)
2. "x can be anything" (Orange_TR)
3. "x can be whatever it wants… but it doesn't matter what x is because it'll still return a value if it's 0 or less than 0." (Orange_TL)
4. "the x can be whatever it wants" (Red_TL)
5. "x is independent of anything, you're just multiplying it" (Yellow_TR)
6. "it's independent of x because x is whatever is returned" (Orange_BR)
7. "it shouldn't matter what x is" (Blue_TL)
8. "x doesn't really matter, so it's n that matters" (Brown_BR)
9. "it doesn't matter what x is… you're not changing x" (Rick: Yellow_BL)

www.manaraa.com

10. “it doesn’t seem that there should be any restrictions on x to return a value”

    (Purple_TR)

Two participants (1 & 2), with almost identical language, claimed that *“x could be anything”* and *“x can be anything.”* These participants provided no justification and made no reference to the line of code responsible, but their statements are very similar to the correct conclusions that x can be any number.

Two other participants (3 & 4) used anthropomorphic language to describe the constraints on the variable x. Both of these participants used the phrase *“whatever it wants”* in reference to the selection of the x value. One of these participants was explicit regarding why the variable x *“can be whatever it wants”* and explained that “*it doesn’t matter what x is because it’ll still return a value if it’s 0 or less than 0.”*

On the other extreme, two participants (5 & 6) used the technical term *”independent”* to describe the relationship of the value of the variable x and whether or not the function will return a value. One participant justified that *“you’re just multiplying it.”* It is true that you are using the value of the variable x in a multiplication operation. However, in terms of justifying the independence of x, the real justification is that x does not determine when the base case has been reached and instead determines the return value, as the participant said, through *“multiplying.”* The second participant that used the phrase *“independent”* justified this claim by saying that *“x is whatever is returned.”* As with the previous participant’s justification that *“you’re just multiplying it,”* this participant doesn’t mention the fact that x does not determine when the base case is reached. The justification that the second participant provides that "*x is whatever is returned”* is both sometimes inaccurate and is irrelevant to whether or not the function produces an infinite loop. First of all, some power of x is returned, which is sometimes, but not always, equivalent to x. Second, whether or not the value of x is returned is irrelevant to whether or not x determines whether the base case can be reached. Despite these two participants’ insufficient justifications, they provide an appropriate technical label for the relationship between the value of the variable x and whether or not the function will return a value.

Three participants (7, 8, & 9) discussed whether the variable x will *“matter.”* One participant asserted that *“it doesn’t matter what x is,”* and also mentioned that *“you’re not changing x.”* This secondary comment would likely be helpful for tracing the value of the variable x, because if “you’re not changing x” it is not necessary to attend to the value of the variable x. The other two participants who used the word “*matter”* sounded less confident in their responses. The first said *“it shouldn’t matter what x is”* and indicated some lack of confidence through his tone and use of the word *“shouldn’t.”* The other participant explained that *“x doesn’t really matter, so it’s n that matters.”* It is accurate that only n matters for whether or not the base case will be reached, but the participant is slightly imprecise with their

www.manaraa.com

language by not clarifying that x is relevant to the output of the function. Again, this participant's tone sounded less confident than the first participant described in this paragraph.

The final quotation (10) is similar to the others, but does not fall into any of the previous clusters of responses. With some hedging language, *"it doesn't seem that,"* the final participant claimed that there shouldn't be *"any restrictions on x to return a value."* The sentiment in this participant's statement is not unique; only the language of *"restrictions"* was unique in the sample of participants' answers to the infinite-loop question.

The generalization regarding the independence of the number of recursive calls made and the value of the variable x could be beneficial to an individual tracing the recursive function because they could use this generalization to focus their attention on the value of the variable n. Tracing recursive functions can require tracking a number of variables simultaneously. Insight regarding the roles of those variables may provide the opportunity to check the tracing of individual variables or to recognize when and why the values of particular variables will be most important for the purpose, narrowing the individual's focus and avoiding distractions.

Conclusions

The above analyses detailed some patterns and subtleties in participants' partial descriptions of state change for the variables x and n. However, this analysis did not answer the open question of what advantage could be provided when tracing the recursive function. I expect that being able to describe the patterns of state change such as the partial descriptions of state change I described here is important to programming competence. Given that participants answered the tracing question before the infinite-loop question, my answer to this question is not justified with data. However, it is possible to justify this expectation even from the fact that experts in the domain of computer science would be capable of generating what I refer to as partial descriptions of state change, including both multiline and single-line summaries.

Beyond that justification, attending to these patterns of state change could be helpful to an individual when tracing a recursive function call. One of the challenges of tracing through recursive functions is tracking the state of variables across each recursive call. Recognizing patterns of state change could provide a resource to check the specific steps when tracing the state of variables in a set of recursive calls or even to avoid duplicate or unnecessary checks. This type of mechanism for checking an individual's detailed tracing of a function may be a significant resource for successfully tracing a recursive function.

The recognition of these patterns of state change can be seen as distinct from the process of tracking specific elements of state. Using an analogy of the mind as a computer, a computer, when executing recursive calls, does not have a mechanism to identify these patterns of state change and it would require an additional mechanism. So too, this requires two paths of reasoning for an individual, which may or may not be an integrated cognitive process for a human. However, building upon the analogy of the mind as a computer we can think of tracking individual elements of state and developing partial descriptions of state change as being two distinct tasks. The first is to trace through individual lines of code and to be

www.manaraa.com

able to determine for any input, the resulting behavior of the function. A second is to be able to describe this pattern of behavior for relevant ranges of input. The infinite-loop question appeared to orient students to noticing details about the function that are at this second level of description.

To validate the hypothesis that multiline summaries support individuals in coordinating state, additional data would be needed. A study could compare individuals' performance on various tracing and infinite-loop questions and vary the ordering of these questions. This could assess if the type of reasoning students used to answer the infinite-loop question is used productively when tracing the question if they first reason about the cases that produce an infinite loop. However, even if this does not spontaneously occur, a teaching study could be used to attempt to elicit this type of transfer. An additional hypothesis to test is that the clinical interview provided support for examples of multiline summaries, but that the same questions outside of an interactional interview would not prompt this reasoning pattern. These forms of investigation could help us develop techniques for scaffolding students in developing multiline summaries and I believe we may be able to achieve more scaffolding than is provided by the prompt “describe in plain English” (Whalley, Robbins, & Lister, 2008; Venables, Tan, & Lister, 2009).

The following chapter will continue to explore participants' competence on the infinite loop question by refining hypotheses regarding what intuitive knowledge about infinite loops and base cases that may have supported this competence. A question that will remain unanswered is why the infinite-loop question elicited this intuitive knowledge while the tracing question did not.

www.manaraa.com

INTUITIVE KNOWLEDGE ABOUT BASE CASES AND INFINITE LOOPS

A body of computer science education research investigates students' understanding of the concept of recursion (Kurland & Pea, 1989; Kahney, 1989; George, 2000; Clancy, 2004). This previous research shows that students experience persistent difficulty understanding the concept of recursion. However, I know of no research that identifies aspects of recursion that are *unproblematic* for students. In addition, I know of no research that analyzes which aspects of recursion are built on, or could be built on, robust intuitive resources. To investigate these undocumented aspects of individuals' understanding of recursion, I analyze students' intuitive resources about infinite loops and base cases from an interview study with introductory programming students. I found that participants' explanation of how to create an infinite loop in a particular recursive function were accurate, used varied non-technical language, and may have built upon intuitive knowledge. This was true even for participants that demonstrated poor performance when attempting to trace the same recursive function. The central question in this chapter is: what prior knowledge accounts for participants' robust reasoning regarding infinite loops?

The analysis in this chapter is governed by the Knowledge in Pieces theoretical framework (diSessa, 1993), which motivates the analytic focus on students' strengths rather than the typical focus on students' weaknesses (Smith, diSessa, and Roschelle, 1993). Smith, diSessa, and Roschelle (1993) critique research focused on identifying misconceptions as characterizing participants' prior knowledge as fundamentally unproductive. They argue for developing more comprehensive models of participants' learning so as to better understand the role of prior knowledge and both students' strengths and weaknesses. The current study is in line with the research direction set out by Smith, diSessa, and Roschelle (1993) and has both theoretical and practical relevance to computer science education. This direction has theoretical relevance; we have only partial understanding of the learning process without exploring the role played by prior knowledge in both successful and unsuccessful learning attempts. This direction also has practical relevance; we may be able to develop pedagogy that capitalizes on participants' intuition and prior knowledge as has been done in other domains (*e.g.,* diSessa & Minstrell, 1998).

The motivation for this and the previous chapter was the fact that students who demonstrate some lack of understanding of recursion were still correct in reasoning about the cases that produce an infinite loop. All of the seventeen students that participated in the study demonstrated correct reasoning and arrived at the correct answer for the infinite-loop question shown in Figure 41. This is surprising and worthy of study because participants' proficient reasoning regarding infinite loops accompanied a variety of levels of overall proficiency with recursion on the tracing question shown in Figure 41. This pattern need not be universal to motivate the current exploration, because it may be possible to support individuals to achieve the same performance. Motivated by these general patterns of participants' correct reasoning, I set out to analyze the nature of the participants' knowledge about infinite loops and base cases.

www.manaraa.com

```
(define (whatIsIt x n)
   (if (= n 1)
        x
       (* x (whatIsIt x (- n 1))) ))
```

What value is returned by `(whatIsIt 4 4)`?
A) 8   B) 16   C) 24   D) 64   E) 256

Figure 40. The "tracing question": a translation of a question from the 1988 APCS exam.

Which of the following is a necessary and sufficient condition for the function `WhatIsIt` to return a value if it is assumed that the values of n and x are small in magnitude and are both whole numbers?
A) $n > 0$
B) $n = 0$
C) $n > 0$ and $x > 0$
D) $x \leq n$ and $n > 0$
E) $n \leq x$ and $n > 0$

Figure 41. The "infinite-loop question": a replication of a question from the 1988 APCS exam.

To develop hypotheses about the nature and origins of participants' knowledge of infinite loops and base cases I build upon both a line of work in cognitive linguistics (Lakoff & Núñez, 2000; Lakoff & Johnson, 1980), which I will refer to as Metaphor Theory, and the Knowledge in Pieces theoretical framework (*e.g.*, diSessa, 1993). The ideas that I build upon from both Metaphor Theory and the Knowledge in Pieces theoretical framework relate to students embodied experience. The research program was designed to identify plausible undocumented potential connections between out-of-domain knowledge and computer science and not to validate a particular connection. Therefore these connections between embodied experience and computer science are at least somewhat speculative and would require additional research to refine or validate. However, with the data available I show that these hypothesized connections are plausible and warrant additional investigation.

Content logs were created of all video data and notes were made regarding segments of interest and possible patterns. With this process, I identified the pattern that participants demonstrated correct reasoning and articulate explanations on the infinite-loop question despite many participants having difficulty tracing the same function.

All participants' solutions to the infinite-loop question were transcribed. From these transcripts and associated video clips, I developed local hypotheses that were refined by considering more data. This was an iterative process informed by the methodology described by Engle, Connant, and Greeno (2007). For example, an initial hypothesis was that participants' competence on the infinite-loop question might be because the function `WhatIsIt` was particularly easy for students to reason about. This was an initial hypothesis that was rejected

www.manaraa.com

because many participants demonstrated incorrect reasoning on the tracing question and correct reasoning on the infinite loop question.

A second initially plausible hypothesis, which was later rejected, related to participants providing memorized responses. I use the phrase “memorized response” to label a situation in which a phrase presented in formal instruction is repeated by a participant without changes or with insignificant changes to wording. In the analysis I provide a representative selection of participants’ statements about infinite loops and base cases to show that participants’ language for describing infinite loops was varied and non-technical. I claim these characteristics are unlikely in a memorized response. This serves as both motivation for the later analysis and as a result documenting a competence of the research participants.

The analysis is broken into hypotheses regarding the nature and origins of participants’ knowledge of infinite loops and base cases.

The first hypothesis explored in this chapter is that an individual’s understanding of infinity or components of that understanding can support their reasoning about infinite loops. I developed this hypothesis from the data by applying both Metaphor Theory and the Knowledge in Pieces theoretical framework. In Metaphor Theory, Lakoff and Núñez (2000) articulate the way in which individuals’ understanding of infinity builds upon their understanding of ongoing iterative processes. I describe this embodied knowledge identified by Lakoff and Núñez (2000) and how it may relate to students’ understanding of infinite loops. I hypothesize that this same resource that Lakoff and Núñez (2000) claim supports an understanding of infinity, individuals’ understanding of ongoing iterative processes, supports participants’ reasoning about infinite loops. From the Knowledge in Pieces perspective I discuss how this intuitive knowledge may be supported by a previously undocumented p-prim that I refer to as the *repeating p-prim.*

I observed that some of the participants’ statements about base cases could be interpreted as using physical language and in this chapter I develop hypotheses about the nature of participants’ knowledge of base cases by applying both Metaphor Theory (Lakoff & Núñez, 2000) and Knowledge in Pieces (diSessa, 1993). My application of metaphor theory suggests that students used two metaphors when describing base cases, and I will refer to these as Base-Case-State-as-a-Destination and Base-Case-State-as-a-Goal. My application of Knowledge in Pieces suggests that students used the *blocking p-prim* to reason about the blocking role of the function’s base case.

My analysis sought to identify potential sources of individuals’ competence on the infinite loop question by analyzing the language participants used to describe infinite loops and base cases. This analysis also highlights potentially rich sources of knowledge that may support pedagogy for teaching recursion. I did not set out to validate these hypotheses, but through developing the hypotheses from the data I have connected computer science education to both the Knowledge in Pieces theoretical framework (diSessa, 1993) and Metaphor Theory (Lakoff & Núñez, 2000). In particular, I provide two contributions to Knowledge in Pieces theory. First, I propose as previously unidentified p-prim that I refer to as the *repeating p-prim*. Second, in developing this p-prim I propose a clarification of diSessa’s claim that p-prims are inarticulate

www.manaraa.com

(1993) and specify that although the use of a p-prim provides an expectation that an explanation is unnecessary, when it is brought to an individual's attention he or she may still be able to generate an explanation or draw a conclusion that an explanation could be provided.

Motivation: No Evidence of the use of a Memorized Response

If students provided memorized answers to the infinite-loop question, it would provide an explanation for the observation that many students answered the infinite-loop question correctly even though they experienced difficulty on the tracing question. For example, I can memorize and recall statements from a variety of domains about which I am ignorant. Responding with this memorized statement in this case would not necessarily indicate an understanding of concepts from that domain.

Such memorized elements may be present in the knowledge system of a student with limited knowledge. However, they may also be present in an expert's knowledge system. For example, an expert physicist certainly has memorized the phrase "F=ma." For an expert physicist, this phrase may be encoded as this specific and very familiar phrase, but the phrase also relates to the expert's understanding of forces, mass, and acceleration. Therefore a "memorized response" is not necessarily a statement devoid of conceptual meaning. However, eliminating a memorized response as the source of participants' explanations implies that participants constructed an explanation based upon their knowledge, which might not be the case when generating a memorized response.

I expected a memorized response to include technical language already introduced in the course. Given that the majority of the participants were enrolled in the same course, I expected that a memorized response used by one participant would show up as repeated sequences of words used by multiple participants. Even if this language were non-technical it might be widely used by students if it was introduced in the course.

I present quotations in which participants discussed infinite loops and base cases when answering the infinite-loop question. The quotations are used to verify that participants did not display evidence of what I defined as a "memorized response," which is a phrase presented during formal instruction that is repeated by a student without changes or with insignificant changes to wording. This would include technical language or systematically repeated phrases among participants. General patterns among these quotations are noted, but are tangential to the primary finding that participants did not appear to use memorized responses. This finding is used here to motivate the further analysis of participants' responses.

Table 2 shows quotations from participants who described an infinite process when answering the infinite-loop question. Participants alluded to the infinite process using everyday words such as *"continues"* (Brown_TL) and only one participant used the technical phrase *"infinite loop"* (Orange_TL). Some students described the unending nature of the process by mentioning that it *"goes on forever"* (Green_TL) or *"forever and ever and ever"* (Green_TR) or by using words such as *"infinitely"* (Orange_BR) and *"infinity"* (Brown_TR). Three students focused on the lack of a stop or end of an infinite loop again without using this technical term.

www.manaraa.com

They said that the *"function never stops"* (Brown_TR), that it is *"never going to end"* (Orange_TL), and that it will keep going *"unless it like stops"* (Purple_TL).

Table 2. Participant descriptions of an infinite process.

- "it just continues being called" (Brown_TL)
- "just keep doing it" (Yellow_TR)
- "it just goes on forever" (Green_TL)
- "the problem will go on forever and ever and ever" (Green_TR)
- "then it will go on infinitely" (Orange_BR)
- "the function never stops , it just repeats itself till infinity" (Brown_TR)
- "it's never going to end; it's going to be an infinite loop." (Orange_TL)
- "this will continue to keep on going unless it like stops" (Purple_TL)
- "then you go down to negative infinity oblivion" (Scratch_Th)

Table 3 shows a collections of quotations in which participants describe the base case when solving the infinite-loop question. There was a variety of language used to describe the base case; for example, participants said that *"n has to be able to become one"* (Orange_TR) and that you *"have to eventually get n equals one"* (Red_TL). Participants also talked about the *"condition"* (Red_TR & Orange_BR) that needed to be *"satisfied"* (Red_TR & Red_TL) or *"fulfilled"* (Orange_BR). Only a single student used the technical language of "base case." Again, there are patterns in the participants' responses, but we can see no systematically repeated phrases or technical language.

Table 3. Participant descriptions of base cases.

- "it equals one and the program stops" (Brown TR)
- "n has to be able to become one in the end" (Orange_TR)
- "you actually have to eventually get n equal to one" (Purple_TR)
- "this will keep on going on until n equals one" (Red_TL)

www.manaraa.com

- “we know that we need to get n equal to one” (Yellow_BR)
- “for your recursion to stop your ending condition could be satisfied” (Red_TR)
- “you want to get to n equals one to satisfy this part” (Red_TL, duplicate words removed)
- “this is the condition that has to be fulfilled” (Orange_BR)
- “this is like your exit function right? (referring to the test n==1)” (Purple_TL)
- “if n is one, n would just come in here (referring to the true case of the conditional)” (Scratch_Th)
- “N must be greater than 0, so that it can return x” (Brown_BR)
- “the base case is n equals one” (Yellow_BL)

The variety of the participants’ language suggests that they had not simply repeated language they had learned in their programming class. It is unlikely that students would have used such a varied set of non-technical language to describe a technical process introduced in their programming course if it was completely unconnected to their prior knowledge and experience. This result that students were unlikely to be providing memorized responses serves as motivation for the following analyses, which discuss hypotheses for the nature of participants’ knowledge about infinite loops and base cases.

Previous Research

Potential Infinity and Actual Infinity

A potentially prerequisite concept for understanding infinite loops is the concept of infinity. There has been extensive developmental research on children’s understanding of infinity (Evans, 1983; Monaghan, 2001; Falk, 2010), which informs my assessment of the plausibility of this hypothesis.

Here I review a recent article that summarizes the development of children’s understanding of infinity (Falk, 2010). Based upon previous research, Falk identifies components of infinity that are more and less understood by individuals. For example, she separates two models of infinity: potential infinity and actual infinity. Potential infinity is identified as a process such as counting that progresses toward infinity, which Falk and others have identified as easier to understand. Falk (2010) summarizes from other studies “that children’s repeated experience of forming successors while counting eventually leads, by

www.manaraa.com

induction, to conceiving the unending succession." (p. 27). The more difficult idea of actual infinity represents infinity as an object, which in reality can never exist.

Falk (2010) summarizes that "roughly from about age 8 on, children grasp potential and actual infinity." (p. 1). This developmental finding removes concern that the population of the current study, college students, would lack rudimentary understanding of infinity. It is important to note that this expectation that participants "grasp" (p. 1, Falk, 2010) infinity does not imply that these ideas will be accessible or productively used by participants in a new context like computer programming; it only implies that a grasp of infinity could serve as a resource.

Falk (2010) and other researchers investigating children's understanding of infinity (*e.g.*, Evans, 1983; Monaghan, 2001) typically use developmental studies and do not develop models regarding the nature of this knowledge. I continue with a variant of the hypothesis that participants' knowledge of infinity contributed to their competence, but turn to research that discusses the nature of this knowledge.

Metaphor Theory and the Basic Metaphor of Infinity

In the development of my hypotheses about the nature of participants' knowledge about infinite loops and base cases, I applied what I refer to as Metaphor Theory. This line of research from cognitive linguistics has had many contributors over the past 30 years. I will ground my description of Metaphor Theory in the work of George Lakoff and colleagues (Lakoff & Johnson, 1980; Lakoff & Núñez, 2000). Lakoff and Núñez (2000), summarizing this lineage of research that I refer to as Metaphor Theory, claim that

> "One of the principal results in cognitive science is that abstract concepts are typically understood, via metaphor, in terms of more concrete concepts. This phenomenon has been studied scientifically for more than two decades and is in general as well established as any result in cognitive science" (p. 40-41, Lakoff & Núñez, 2000).

Lakoff and colleagues (*e.g.* Lakoff & Núñez, 2000; Lakoff & Johnson, 1980) map the language individuals frequently use when describing or discussing a topic to a metaphor or set of metaphors that make that use of language intelligible. For example, they provide examples in which affection is "understood in terms of physical warmth" (p. 41, Lakoff & Núñez, 2000). Using words like "warm," "cold," "icy," and "ice" in a sentence about affection makes use of this metaphor. These examples of language are assumed to be comprehensible only through the unconscious interpretive lens of this metaphor.

A central claim of Lakoff and Núñez (2000) is that these metaphors are embodied, or developed through physical experience. They explain how the concept of infinite processes and what they call the Basic Metaphor of Infinity (BMI) is developed despite the fact that no embodied experiences are truly infinite. The central claim underlying the Basic Metaphor of Infinity is that indefinite processes are conceptualized as iterative processes, which are processes that repeat. For example, counting can be conceptualized as an unending process.

www.manaraa.com

Lakoff and Núñez (2000) argue that the source of knowledge about infinite processes is rooted in this aspectual system. Lakoff and Núñez (2000) identify the aspectual system as the area of the brain that processes the grammatical aspect of verbs. Grammatical aspect, distinct from tense, includes information regarding time, such as the duration of an action, the completion of an action, and the frequency of an action.

Lakoff and Núñez (2000) make the frequently contested argument "that a considerable number of infinite processes in mathematics are special cases of the BMI that can be arrived at by specifying what the iterative process is in detail." (p. 161, Lakoff & Núñez, 2000). While these claims regarding the application of the Basic Metaphor of Infinity to various topics have been contested (*e.g.*, Schiralli & Sinclair, 2003), I know of no critics who challenge the claim that individuals' understanding of iterative processes could serve as a resource for other understanding, which is the aspect of the theory that is relevant to the current analysis.

Hypotheses Regarding Infinite Loop Knowledge

Relevance of Actual and Potential Infinity

I believe that actual infinity is not relevant to understanding participants' reasoning about infinite loops in recursive functions because the `WhatIsIt` function would not create a loop that could continue forever. Many of the students mentioned that the function would "crash." This is a correct prediction for the `WhatIsIt` function, which in the case of an infinite loop would eventually fill up all available memory on the computer. This practical reality appears to eliminate the need for an understanding of actual infinity, which is the idea of infinity as an object, for understanding infinite loops in the `WhatIsIt` function. However, potential infinity (Falk, 2010), which is the idea of an unending succession, is a relevant idea for reasoning about infinite loops because infinite loops are often created through repeated execution of a function. Therefore, an individual's understanding of potential infinity is relevant, but not an individual's understanding of actual infinity.

Relevance of Iterative Processes

Lakoff and Núñez (2000) claim that individuals conceptualize many infinite processes (which by definition have no completion or result) as if they were an iterative process with an intermediate result. However, an infinite loop *is* an iterative process with intermediate results. Lakoff and Núñez (2000) explain that the intuitive roots of the Basic Metaphor of Infinity are an understanding of iterative processes and therefore the intuitive roots of the Basic Metaphor of Infinity are a plausible contributor to participants' reasoning.

The Hypothesized Repeating P-prim

diSessa (1993) developed a theory about a type of intuitive knowledge, which is frequently derived from physical experience. He referred to this intuitive knowledge as a phenomenological primitive or a p-prim for short. The theoretical framework chapter provides a more extensive description of p-prims. Here I build upon this theoretical framework to identify a p-prim that describes intuitive competence comparable to the experience with iterative processes discussed by Lakoff and Núñez (2000). I present this previously unidentified p-prim, the *repeating p-prim,* and how it connects to competence with infinite loops.

www.manaraa.com

The Repeating P-prim

Schematization: An identifiable pattern of behavior in or of a system is performed multiple times in sequence. The repetition is rhythmic and predictable, not random or unpredictable.

Attributes: An identifiable behavior, regularity

Relation to schooled physics: The earth repeats its motion around the sun, the earth spins on its axis, and the moon revolves around the earth. Individuals may see these behaviors as repeating a specific cycle and not as a dynamic process of forces and momentum producing a particular pattern of behavior. Another example is a heartbeat. When individuals think about a heartbeat, they are unlikely to draw upon the complex fluid mechanics knowledge that would be necessary to truly explain the phenomenon. Instead, a heartbeat can be conceptualized as a rhythmic and predictable repetition.

Comments: The *repeating* p-prim is also present in everyday physical experiences such as walking and breathing. These both constitute identifiable patterns that are conceptualized as a single behavior that is repeated and are, without question, part of everyday experience. The *repeating* p-prim can also be seen in visible artifacts, such as the white lane lines on a freeway, which reoccur visually as you drive past.

Proposed Modification to P-prim Theory

P-prims provide a sense of obviousness for a phenomenon and with this sense of obviousness individuals can view the phenomenon as not needing an explanation. diSessa (1993) models p-prims as inarticulate, which means this sense of obviousness is not provided by an articulate description of the situation. The repeating p-prim initially appears to violate the methodological heuristic for determining p-prims in that they are not articulate (diSessa, 1993). I will provide an explanation for why I believe that this property of the repeating p-prim is likely to be true in some cases for other p-prims.

I propose a clarification to the claim made by diSessa (1993) that p-prims are inarticulate. I do not challenge or attempt to amend that p-prims describe a range of phenomenon and do so without the use of language. I hypothesize that with directed attention an individual may override the view of the phenomenon as not requiring an explanation. I propose that if the need for an explanation is brought to his or her attention, in some cases an individual may also be able to articulate what they believe to be an explanation of the particular phenomenon or articulate that an explanation could hypothetically be provided.

Sometimes, but not always, this explanation may be a scientifically accepted explanation. For example, consider a physicist interacting with the world around her. This physicist is capable of providing a scientifically-normative explanation for how a book rests on a table. However, when moving objects around a desk, she does not need to reason about the placement of each book starting from first principles. Instead, she can rely on more primitive physical intuition such as the supporting p-prim. The supporting p-prim is schematized as "'Strong' or stable underlying objects keeps overlaying and touching object in place." (p. 220, diSessa, 1993). Her interaction with the physical environment can be guided by this inarticulate

www.manaraa.com

p-prim and not her full explanation. It would be immensely inefficient if she always reasoned from first principles that the book will rest on the table. Again, this does not imply that the p-prim itself is articulate, only that there can exist, in parallel to the competence of applying the p-prim, a competence to provide a correct scientific explanation.

P-prims vary in the level of knowledge necessary to be able to provide what the individual believes to be an explanation for the phenomenon. The example above described an individual that could provide a scientifically-normative explanation, but an articulate explanation need not be scientifically normative to be an instance where an individual believes an explanation to be necessary. For example, the dying away p-prim explains why an object that is pushed will eventually come to a stop. When applying the dying away p-prim the individual sees the phenomenon as requiring no explanation. However, if the need for an explanation is brought to their attention, many non-physics experts may believe that they can provide an explanation of this phenomenon. This may appear to violate diSessa's claim that p-prims are inarticulate (1993), but I propose that it is not necessary to assume that an individual could not produce an articulate explanation for the phenomenon addressed by the p-prim.

Many situations in which the repeating p-prim can be applied may be seen as explainable. Consider the application of the repeating p-prim to understanding walking. I expect that in most cases individuals view walking as requiring no explanation, but if the need for an explanation is brought to their attention they may be able to provide what they believe to be an explanation of the phenomenon.

In the example of the physicist, her hypothetical p-prim use when moving objects on a desk appeared disconnected from her scientific knowledge. The question remains whether the hypothesized *repeating p-prim* provides a similar type of inarticulate intuition for a class of phenomenon. Is it possible for an individual to use only the *repeating p-prim* to reason about a particular repeating phenomena and not their articulate knowledge? If this is not possible for a particular phenomenon, it implies that the repeating p-prim does not apply to the phenomenon. If the hypothesized repeating p-prim provides only articulate reasoning for all relevant phenomena it would not be classified as a p-prim.

Hypotheses Regarding Base Case Knowledge

During the analysis, I observed that participants used metaphoric language in their explanations of base cases, which informed an initial hypothesis that metaphor, or the source domain of these metaphors, provided resources that produced the competent performance observed on the infinite-loop question. I developed and present two hypothesized metaphors used by the participants. I have named these two metaphors, Base-Case-State-is-a-Destination and Base-Case-State-is-a-Goal, using the naming conventions used by Lakoff and Núñez (2000). I provide brief quotations from the participants' responses to the infinite loop question to demonstrate some of the use of metaphoric language specifically aligned with each of these metaphors. I hypothesize that the *blocking p-prim* (diSessa, 1993) may be productive for reasoning about base cases and that participants' use of physical language may be evidence of the use of the blocking p-prim.

www.manaraa.com

The first hypothesized metaphor is referred to as Base-Case-State-is-a-Destination. The participants sometimes discussed the state that satisfies the base case as if it were a physical location or destination. The metaphoric language in these examples suggests a physical arrival at a stopping condition. For example, the variable n was described as needing to have *"got to"* (Purple_TL) or to *"reach"* (Purple_TR) a location or state. Another participant described the infinite loop case as *"it will never hit an end to the recursion."* (Purple_TR). The language of *"hit"* suggests that the *"end to the recursion"* is a physical location that can be *"hit."* Another participant explained the non-infinite loop case as *"it will eventually hit n equals 1 at one point"* (Red_TL). This participant's statement included a similar use of the word *"hit"* and the phrase *"at one point,"* which suggests the passage of time and is consistent with a metaphor of reaching a physical location.

The second hypothesized metaphor will be referred to as Base-Case-State-is-a-Goal. There was another set of participants that described the state that satisfies the base case as a goal without the physical language shown above. The following examples are consistent with a non-physical specification of the word goal. Participants mentioned that you *"have to have n equal to 1"* (Red_TR), *"have to eventually get n equal to 1"* (Purple_TR), and that n *"has to be able to become one in the end"* (Orange_TR). In addition to these examples that emphasized the need or goal for the value of the variable to equal one, other participants described this goal as something that the code needed to *"satisfy"* or *"fulfill."* For example, participants explained that the base case condition *"has to be fulfilled"* (Orange_BR), and that *"you want to get to n equals one to satisfy this"* (Red_TL), and that *"we need to get n equal to one"* (Yellow_BR).

I do not have evidence from my study that metaphor functioned to determine individuals' reasoning. However, given the observation of metaphoric language, metaphor may have contributed to individuals' reasoning. From the Knowledge in Pieces theoretical framework, I assume that individuals may appear inconsistent in their reasoning and therefore may be inconsistent in their metaphor use. This challenges the idea that metaphor shapes individuals' reasoning patterns. I expect that there is diversity in individuals' cognitive resources and that metaphor may enable the application of cognitive resources that might not otherwise be applied. For example, in the case of the Base-Case-is-a-Destination metaphor, an individual's knowledge and experience of destinations may support reasoning about base cases. Perhaps even this use of metaphoric language suggests that this could be a productive anchor for students' understanding of base cases.

Based upon analyzing my data using ideas from the Metaphor Theory, I conclude that students can use metaphoric language to describe the state that satisfies the base case. However, I cannot claim that a single metaphor or even a set of metaphors is responsible for the competence observed because in the data corpus not all students used a single metaphor and individual students used metaphoric language intermittently. Because Lakoff and Núñez (2000) present hypothetical instances of metaphor use, and not from naturally occurring speech, I do not have a reference for what level of uncertainty should be expected in participants' statements.

www.manaraa.com

If we were able to identify a single metaphor that was responsible for the competence with base cases, say the Base-Case-is-a-Destination metaphor, pedagogically relevant questions would remain with regards to the nature of individuals' knowledge about destinations. What knowledge and types of knowledge about destinations are utilized when using the metaphor? How do differences in individual's knowledge about destinations shape their use or understanding of this metaphor? There may be a p-prim, which is a description of physical intuition built by experience, which contributed to reasoning about base cases and can shed light on the nature of participants' embodied knowledge.

diSessa (1993) schematized the *blocking p-prim* as explaining when "an object's tendency toward motion is thwarted by another object in its path" (p. 133, diSessa, 1993). This can be seen as a rough approximation of the role of a base case in preventing continued execution of a recursive function providing correct intuition. I hypothesize that the *blocking p-prim* may be productive for reasoning about base cases and that participants' use of physical language may be evidence of the use of the blocking p-prim.

diSessa explains that the *blocking p-prim* does not imply agency. "Blocking (what a heavy brick does to a hand striking it) and bouncing impute no agency, but are kinematic, as it were, describing phenomena visually and geometrically." (p. 128, diSessa, 1993). This lack of agency is consistent with the participants' statements about base cases because only the variable n and the recursive function were described as actors and never the base case. Instead, the base case was something that you *"got to"* (Purple_TL), could *"reach"* (Purple_TR) or even "*hit"* (Purple_TR). These descriptions appear to be spatial, which is consistent with the genesis of the *blocking p-prim* in physical experience.

diSessa describes that the development of physics mastery requires decomposing the behavior described by the *blocking p-prim* into relevant forces and that the *blocking p-prim* alone provides a naïve and not scientific explanation of some physical situations. Similarly, in the computer science context, the *blocking p-prim* is relevant to how the recursion stops, which is actually through the absence of the recursion being continued by a recursive call and not through blocking per se.

Conclusion

Recursion is typically identified as one of the most difficult concepts in computer programming and novice programmers frequently make mistakes in writing recursive functions, which sometimes generate infinite loops. The data presented in this chapter contrast the community's perception of the difficulty of recursion by identifying an area of strong intuitive knowledge, which supports correct reasoning about some aspects of recursive processes. I argue that some components of infinite loops are "easy" and I have demonstrated what may be some substantial strength for reasoning about infinite loops in recursive functions. From a theoretical perspective, this is important progress toward the goal of developing a theory regarding the nature of programming knowledge and knowledge of recursion more specifically. This is not intended as an argument that competence with recursion is in fact easy to acquire, only to present hypotheses about specific intuitive knowledge on which additional competence

www.manaraa.com

could be built.

I documented participants' varied and non-technical language to describe infinite loops and base cases. I claimed that because of these features participants were unlikely to be simply repeating memorized phrases from instruction when answering the infinite-loop question. This motivated my investigation of the nature of the knowledge that participants used in this context.

I concluded that individuals' understanding of infinity, particularly some of the problematic aspects of this understanding, were unlikely to be necessary for reasoning about infinite loops. Instead, embodied knowledge that Lakoff and Núñez (2000) identified as supporting individuals' understanding of infinity can be seen as relevant to understanding infinite loops. Lakoff and Núñez (2000) characterize this knowledge as derived from the aspectual system. To characterize the same knowledge I document what I believe to be relevant and previously undocumented p-prim, the *repeating p-prim.* In introducing this p-prim, I offer up a refinement to the model of p-prims, which is, in essence, that p-prims can exist in parallel with articulate explanations of the same phenomenon.

Lastly, I documented that participants' language to describe base cases used physical and metaphoric language. This could be evidence of the use of metaphor and I developed two potential metaphors that may have been used by participants. Building upon a similar direction, I proposed that the *blocking p-prim* (diSessa, 1993) may have provided a source of intuition for reasoning about base cases in recursive functions.

The work presented in this chapter sought to identify potential sources of intuitive knowledge with which to reason about infinite loops and not to validate a particular set of hypotheses. This chapter also connects computer science education research with other studies investigating the nature and sources of prior knowledge (diSessa, 1993; diSessa & Sherin, 1998; Lakoff & Núñez, 2000) and also highlights potentially rich sources of knowledge that may support pedagogy for teaching recursion.

www.manaraa.com

SUBSTITUTION TECHNIQUES

The overarching goal of this research program is to better understand students' out-of-domain knowledge that is relevant to reasoning about computer programs. This chapter describes some connections between algebraic substitution and techniques that are applicable to reasoning about recursive functions in a computer science context.

This connection between mathematical substitution and tracing recursive functions became a central focus when the participant Emily[5] (participant identifier: TS_6) connected a technique she used to trace a recursive function with substitution in mathematics. She created the representation shown in Figure 42 and made the following comment, which is relevant to all of the substitution techniques described in this chapter.

> *"Again like I was saying with the math. Math and then you just substitute in something for its equivalent value. Like if they tell you like y is equal to 5 (writes 'y=5') and then you see like 4 times y (writes '4*y') Well you just have to do 4 times 5 (writes '4*5')."*

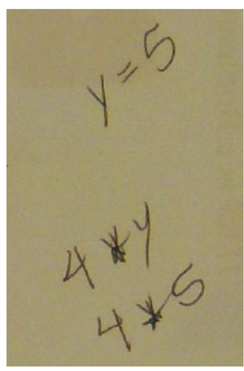


Figure 42 Emily's representation of the connection between substitution in math and programming.

In Emily's example the variable y is replaced with the value of 5. Emily described that you can *"substitute in something for its equivalent value,"* which is a central component for each of the substitution techniques described in this chapter.

The primary audience for these descriptions is computer science educators. This chapter includes descriptions of four substitution techniques that I refer to as *simulating execution*, *accumulating pending calculations, memoization*, and *solving it by hand*. The purpose of this chapter is to provide a clear articulation and prototypical examples of each substitution technique. These techniques were identified within the data corpus of this dissertation and examples of students' reasoning are narrated to provide an example of each technique. The examples show the techniques applied to linear recursion in a functional programming environment. I do not claim that these techniques describe all of the ways in which substitution can be used within computer science. This preliminary taxonomy is speculative and deserves

[5] All names are pseudonyms

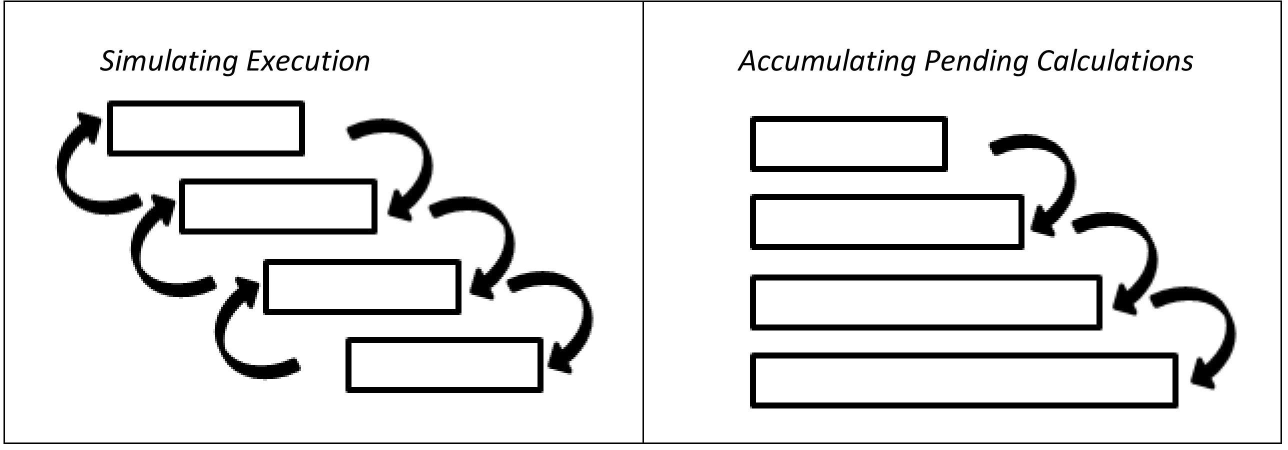
www.manaraa.com

additional research and refinement. I will describe a program of future research. Even in the current form, I hypothesize that these substitution techniques are pedagogically valuable and discuss potential benefits in the discussion section.

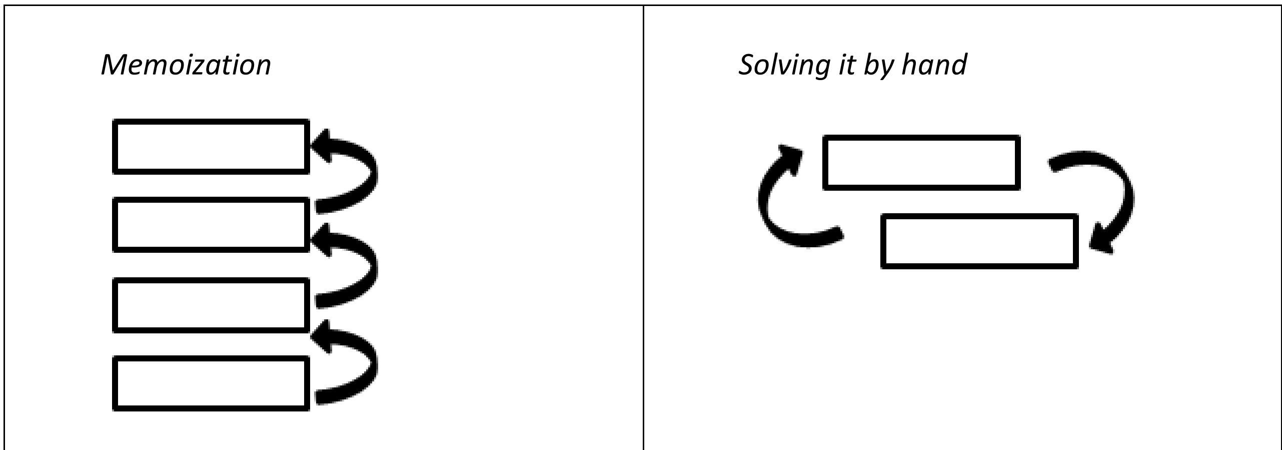
To aid in my description of these substitution techniques I use a typical definition and define a “recursive call” as a function call, within the body of a function, to that same function. The initial function call, which does not originate from within that function, does not constitute a recursive call; I will refer to that as the “initial function call.” Recursive calls are still function calls and I will refer to both initial functions and recursive calls as “function calls.”

Table 4 shows diagrams of the four substitution techniques discussed in this chapter. The top rectangle in each diagram represents the function call that is being traced by the person using this technique. The other rectangles represent recursive calls and, in some of the diagrams, include calculations that are generated during execution of that recursive call. The arrows indicate the order in which the technique requires the individual to reason about each recursive call. Each of these techniques and accompanying diagrams will be described in detail, but even without these details it is possible to observe differences between the techniques in the relative order of execution. The first technique, *simulating execution,* begins at the initial function call and then progresses down to the base case and back up. The second substitution technique is *accumulating pending calculations,* which progresses only from the initial function call down to the base case. The third substitution technique is *memoization,* which progresses only from the base case up to the initial function call. The fourth substitution technique, *solving it by hand,* includes only the initial function call and the first recursive call, but does not consider other recursive calls or the base case.

Table 4. Table of all substitution techniques



www.manaraa.com



Leron and Zazkis (1986) also distinguished different orderings in which recursive functions could be considered. They generalized that mathematicians and computer scientists discuss recursive process as progressing in different directions. They claimed that "mathematicians think of the first part of the definition as a 'start rule', whereas computer scientists refer to it as a 'stop rule'." (p. 25, Leron & Zazkis, 1986) They provided a "likely" description of the factorial function from the perspective of both a mathematician and computer scientist. They claimed that a mathematician would justify that the "definition enables us to compute 1!, then 2!, then 3! And so on to any desired n" (p. 25, Leron & Zazkis, 1986) whereas a computer scientist would justify that "we can compute n! as soon as we know (n-1)! , which in turn can be computed of we know (n-2)! , and so on until we reach 1!" (p. 26, Leron & Zazkis, 1986). In regards to execution order, the mathematician justification is most similar to the substitution technique of memoization while the computer scientist's hypothetical justification is most similar to the *substitution technique of simulating execution.*

Leron and Zazkis (1986) discussed the similarity between recursion and mathematical induction. This is another possible connection between mathematics and computer science, but induction may be no less difficult for students than recursion. This is in stark contrast to the pedagogical recommendation of this chapter to build upon students' competence with algebraic substitution, which I expect is unproblematic technique for many students.

This chapter is inspired by the assumption of college students' competence with mathematical substitution, but I do not develop a nuanced distinction between algebraic substitution and other related techniques within mathematics. Instead I use Emily's example here as a prototypical example of what I believe to be a familiar and common process in algebraic reasoning.

In this chapter I attempt to specify different instantiations of this idea of "substitute in something for its equivalent value" in the context of recursive functions. This set of substitutions may also be helpful for educators to provide students with specific techniques to trace or reason about state in recursive functions. Using more detailed analysis techniques such

www.manaraa.com

as coordination class theory (diSessa & Sherin, 1998) in future work could help develop possible dependencies of applying these techniques.

Methods

The set of substitution techniques was developed from observing the ways participants traced recursive functions. I was interested in the ways in which participants used substitution techniques to reduce the difficulty of tracing a recursive function that relied on principles of mathematical substitution.

The cases were selected to attempt to capture clear examples of the substitution techniques. Since this work is preliminary the cases are used as an existence proof of these techniques and are not analyzed in depth.

This chapter narrates participants' solutions from two of the interview problems. These narrations include my interpretation of participants' statements and details of their apparent use of the substitution technique. These problems included the `WhatIsIt` and `Mult` recursive functions shown in Figure 43 and Figure 44 respectively. A full description of these functions and the questions in which they are found is located within the methods chapter.

```
What value is returned by WhatIsIt(4, 4)?
 (define (WhatIsIt x n)
      (if (= n 1)
            x
            (* x (WhatIsIt x (- n 1)))))
a) 8    b) 16    c) 24    d) 64    e) 256
```

Figure 43. The WhatIsIt Question, a replication of a question from the 1988 APCS exam, translated to Scheme.

Consider the following function where

- x is an integer and x > 0
- It should calculate x * y

```
(define (mult x y)
   (if (= x 1)
       ;; statement 1
       ;; statement 2
   ))
```

Which of the following statement pairs properly completes the function?

```
      <Statement 1>            <Statement 2>
A.      (* x y)                ;; none
B.      y               (mult (- x 1) (+ y 1))
C.      y               (mult (- x 1) (+ y y))
D.      y               (+ y (mult (- x 1) y))
E.      y               (* y (mult (- x 1) y))
```

Figure 44. Reproduced version of the multiplication question from the 1988 APCS exam.

In my explanations of the substitution techniques I will use the factorial function shown in Figure 45, which calculates the factorial for an input x in the programming language Scheme.

www.manaraa.com

```
(define (fact x)
     (if (<= x 1)
         1
         (* x (fact (- x 1)))))
```

Figure 45. Example factorial function written in Scheme.

Substitution Technique: Simulating Execution

Description

I refer to the first substitution technique as *simulating execution*. This is the traditional method of tracing recursive functions whereby the recursive calls are traced in the order they would be executed by a computer. The output from each recursive call is then substituted into the expression that generated that recursive call.

For example, using the substitution technique of *simulating execution* to trace the function fact with the argument 4 would generate the recursive calls shown in Figure 46.

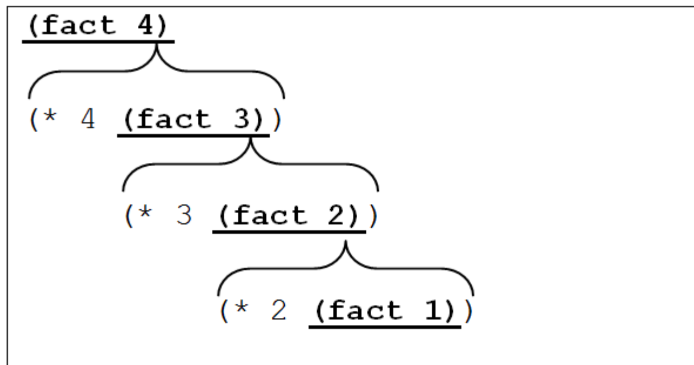


Figure 46. Recursive calls generated by a call to (fact 4).

The underlined calls in Figure 46 are expanded from the top to the bottom. When the base case is reached, the value of 1 is substituted for the call `(fact 1)`. This is multiplied by 2 and the resulting value of 2 is substituted for the call `(fact 2)`. This is multiplied by 3 and the resulting value of 6 is substituted for the call `(fact 3)`. This is multiplied by 4 and the resulting value of 24 is substituted for the call `(fact 4)`.

Figure 47 shows my diagram of this technique. I will describe how each element of the diagram relates to the execution of recursive functions, but I do not assume that students using the technique will necessarily make the same set of connections. Each rectangle represents a function call. The rectangle shown on the top is the initial function call. The arrows to the right of these rectangles represent the instantiation of a recursive call. These arrows show the flow of control in a recursive function, which pauses execution within a particular function call when a recursive function call is made. In a final recursive call, corresponding to the base case where no additional recursive calls are made, the value returned by this recursive call is provided to the calling function that had paused execution. The substitution of this return value at each

www.manaraa.com

step is shown with the arrows on the left of the rectangles. This also represents a change in what code is actively executed. An arrow indicates re-initiating execution, where pending calculation may be executed. Therefore the flow of control begins at the initial function call and then proceeds to each subsequent recursive call before eventually returning from each recursive call in sequence. Each arrow is essentially an instance of substitution; the downward arrows are substitutions that work as an expansion of a particular recursive call and the upward arrows are substitutions of return values from a recursive call. This representation is the most accurate in simulating the flow of control in a recursive function, because each time a value is substituted it corresponds to returning the flow of control to the stack frame for that previous call.

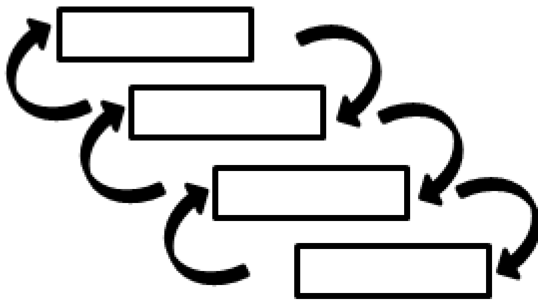


Figure 47. Diagram of the *simulating execution* substitution technique

Example

Figure 48 shows a representation created by a participant (participant identifier: Orange_TR) when tracing the call (`WhatIsIt 4 4`). This example was selected from apparent instances of simulating execution because it showed the most legible and most easily interpreted representation. Each line in her representation shows the expression that would be generated by the recursive call on the previous line. However, she did not show the initial function call (`WhatIsIt 4 4`). When the return value for each line is identified, starting from the bottom, this value can be substituted in the previous line. The participant did not identify each substitution, but summarized "and then you multiply all the fours," which is consistent with the implied substitution in the representation.

* 4 (whatIsIt 4 3)
4 WhatIsit 4 2
4 Whatisit 4 1
4

Figure 48. Written work on the WhatIsIt question by a participant (participant identifier: Orange_TR)

This substitution technique is valid for tracing embedded recursion, where the paused function has pending calculations to be executed when the flow of control returns.

www.manaraa.com

Substitution Technique: Accumulating Pending Calculations

Description

This technique contrasts with the previous in that the calculations that are performed when returning control to a paused recursive function call are accumulated in a single expression containing all pending calculations.

For example, using the substitution technique of accumulating pending calculations to trace the function fact with the argument 4 would generate the recursive calls and calculations shown in Figure 49.

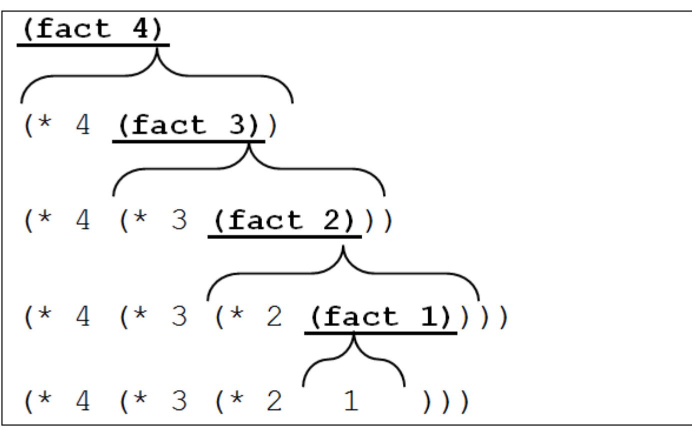


Figure 49. Recursive calls generated by a call to (fact 4).

Again the underlined calls to fact in Figure 49 are expanded from the top to the bottom. However, each line includes all pending calculations. For example, the expanded version of `(fact 3)` is substituted in to the expression `(* 4 (fact 3))` to produce `(* 4 (*3 (fact 2)))`, which is shown on the third line in Figure 49. The same process generates the fourth line. Between the fourth and fifth lines the value returned by the call `(fact 1)` is substituted into the expression to produce the final expression `(* 4 (* 3 (* 2 1)))`. With this substitution technique consideration never returns to previous lines because the final expression contains all necessary state.

Figure 50 shows my diagram of this technique, which unlike the diagram of *simulating execution* in Figure 47 does not include an arrow indicating the flow of control returning to the calling recursive function. Each rectangle still includes a function call, but each rectangle also includes all pending calculations. In the subsequent line, the recursive call from the previous line is replaced with the equivalent expanded recursive relationship. Instead of representing the flow of control, each downward arrow signifies a substitution in which the recursive call is expanded and substituted in to the expression.

www.manaraa.com

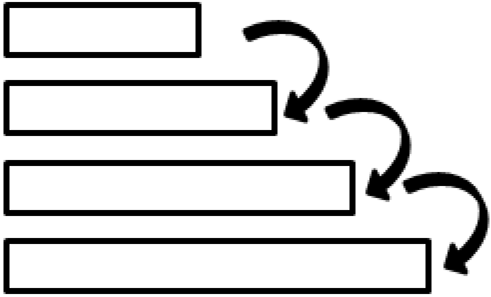


Figure 50. Schematization of the *Accumulating Pending Calculations* substitution technique

The accumulating pending calculations technique does not involve retracing through the previous recursive calls like the substitution technique of *simulating execution* because all pending calculations are accumulated in the final expression.

Example

In the next case, the participant (participant identifier: Purple_Scheme) copied the full expression each time that he substituted in an expanded expression generated by a recursive call. For example, between the first and second line he appeared to substitute "`(* 4 (WhatIsIt 4 2))`" in for the expression "`(WhatIsIt 4 3)`." He was not explicit about this process of substitution, but he arrived at the correct answer and I infer from the representation in Figure 51 that he substituted in the expanded expression from each recursive call.

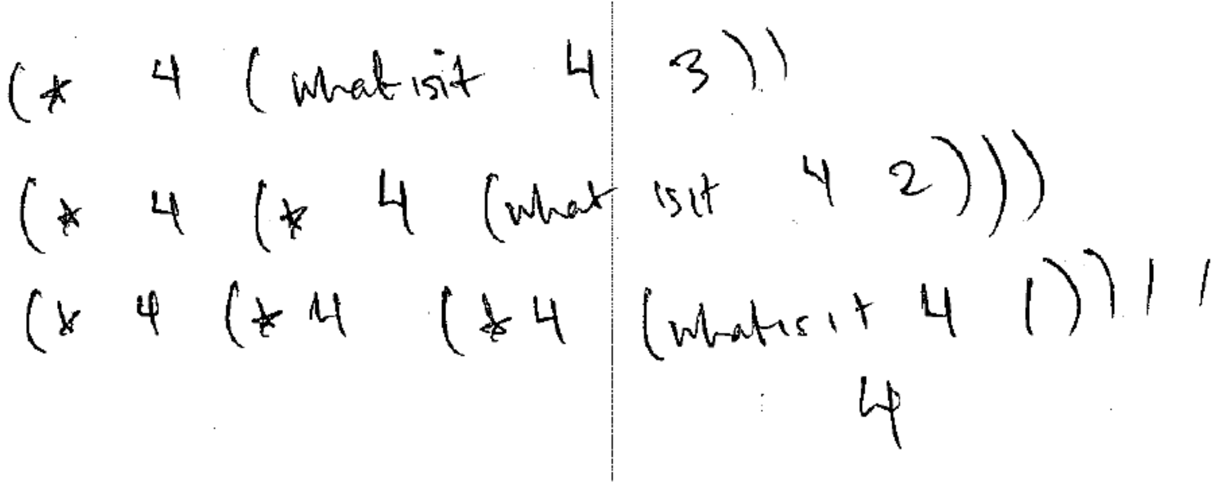


Figure 51. Written work on the WhatIsIt question by a participant (participant identifier: Purple_Scheme)

Like the simulating execution representation shown in Figure 48, he did not write the initial call of "`(WhatIsIt 4 4)`." In the final line in Figure 51, he wrote the number 4, which he said `(WhatIsIt 4 1)` will "return." Writing only 4 on the last line instead of the full set of pending calculations, `(* 4 (* 4 (* 4 4 ) ) )` is a departure from this technique. Despite these subtle departures, this was the most legible and most easily interpreted use of this technique.

www.manaraa.com

Substitution Technique: Memoization

Description

The third abstraction technique relies on calculating and storing the values of particular functions calls before they would be executed by an initial function call. This is similar to the optimization of storing the result of previously calculated recursive calls for the purpose of avoiding redundant function calls. This optimization is referred to as memoization in computer science and because of the similarity to this optimization I refer to this substitution technique as memoization.

Although the technique relies on previously calculated values, this substitution technique can be used to calculate the value for an arbitrary function call. To do this you begin by calculating the value of the recursive function for an input that does not require any recursive calls. In the case of the `fact` function, shown in Figure 45, this would be evaluating the `fact` function with an x value of 1. A call to `fact` with an x value of 1 results in evaluating the true case of the "if" statement and returns the value 1. Now we know that `(fact 1)` returns 1. Next, you evaluate the function call of `fact` with an x value of 2 or `(fact 2)`. This calculation results in multiplying the x value, 2, by `(fact 1)`. We know that (fact 1) returns 1 and can substitute in that value for `(fact 1)`. It would not be an example of the substitution technique of memoization if an individual instead traced the function again for the x value of 1. Now we know that `(fact 2)` returns 2 and this resulting value from `(fact 2)` can be used when evaluating the `fact` function for the value 3. This pattern can be continued to identify the result of an arbitrary recursive call.

This process can be seen as starting at the base case and working toward the desired recursive call. Figure 52 shows a schematization of this substitution technique. For consistency with my diagrams of the other substitution techniques, I have shown the base case at the bottom of this diagram, but this diagram is not representative of the diagrams I would expect individuals to generate. In this substitution technique, the individuals' consideration of the function begins with the base case. If this was written at the top of the individuals' representation, it would generate a diagram that is an inverted version of the one shown.

In Figure 52 the bottom rectangle is a statement of the output of the function at the base case. For an instance of the `mult` function this would be "`(mult 1 5) = 5`." All other rectangles include an expansion of the recursive relationship, such as "`(mult 2 5) = (+ 5 (mult 1 5))`." The arrows show the process of substituting in a previously calculated value such as "`(mult 1 5) = 5`" into the expression above. After this substitution, the full contents of the rectangle would be "`(mult 2 5) = (+ 5 (mult 1 5)) = (+ 5 5) = 10`." Again, the arrows do not show the flow of control, but they show the steps of substitution of previously calculated values such as "`(mult 1 5) = 5`" or "`(mult 2 5) = 10`" into a recursive call that is farther from the base case.

www.manaraa.com

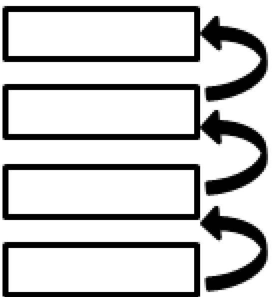


Figure 52. Schematization of the *memoization* substitution technique

Example

Emily, who was quoted as connecting her technique to algebraic substitution, used the substitution technique of memoization on the `WhatIsIt` problem. I will narrate a single step in her use of the strategy.

Emily was reasoning about the expression she had written that is shown in Figure 53. She had already calculated the result of `(WhatIsIt 4 1)` to be 4. In the following transcript, Emily was able to articulate how you could use the result from `(WhatIsIt 4 1)` when calculating `(WhatIsIt 4 2)`. *"I'm thinking that because we found that it was a 4 here, that it would be 4 times 4. And that would be 16."* She then paused and said *"But I think I'm over simplifying things."* I asked her to clarify and she said:

> *"Um because like when it was 4 and 1 like okay, so that was straight forward, but for when it was 4 and 2, what I was doing was like okay, If you have, when you start here. It becomes 4 and 2 minus 1, so then it's 1. So then uh oh well so we know what that is, and that was [4], so then you take it times 4."*

Figure 53. Previously generated tracing of (WhatIsIt 4 2)

A key element in Emily's explanation of this process is her statement *"we know what this is."* This is the central idea in the substitution technique of memoization. Her statement *"that was 4"* stands in place of where she otherwise would have needed to explicitly trace the value of `(WhatIsIt 4 1)`. Emily proceeded to use the same technique to determine the return value of `(WhatIsIt 4 4)`.

Substitution Technique: Solving it by hand

Description

In the substitution technique of *solving it by hand* the individual predicts the output of the first recursive call made in the body of the initial call to the function. I define "predicting the

www.manaraa.com

output of the first recursive call" as using a method that is independent of the recursive function to predict the output of the function. I refer to this substitution technique as *solving it by hand* because the output of the first recursive call is determined "by hand" and not by using the recursive function.

If we were executing the correct version of the function `mult` with the arguments 5 and 3, the first recursive call made would be `(mult 4 3)`. This substitution technique involves predicting the output of `(mult 4 3)`. The function `mult` is supposed to multiply its arguments. Therefore solving it by hand is trivial and `(mult 4 3)` is expected to output the value 12, 4x3. This expected output can be substituted into the expression in place of the first recursive call.

This requires that the individual is able to predict the output of the function and therefore requires that the specification of the function is well understood. The `WhatIsIt` function, for which the behavior of the function is not provided, is not a candidate for the use of this technique.

This technique does not require tracing each recursive call. The initial function call is traced, but then the next recursive call is replaced with a value calculated by hand. This single step of execution is shown in Figure 54, where each rectangle represents a recursive call. The right arrow shows the flow of control that causes the first expansion of the recursive call, but the flow of control to subsequent recursive calls is not shown or considered by the individual using this technique. In the second rectangle, the value that is calculated by hand is substituted into the recursive expression. The left arrow shows the resulting value from this expression returned as the output of the function.

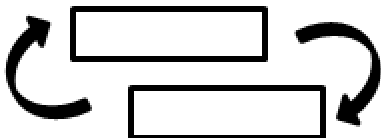


Figure 54. Schematization of the *solving it by hand* substitution technique

This is a normative technique to evaluate the correctness of a recursive call, which is parallel to checking one case of an inductive chain. However, this technique does not guarantee that the recursive function is correct; the base case also needs to be correct and the recursive relationship needs to be consistent throughout the execution of the recursive function.

Example

In the following examples, the participant Tim (participant identifier: Orange_TL) used this technique twice when reasoning about the recursive function `mult` as specified by two answer options. Tim had already used the substitution technique of *simulating execution* to

www.manaraa.com

trace the function, but because of a systematic error in his tracing of answer option C, he could not distinguish the behavior of the functions specified by answer options C and D. The recursive relationship for these answer options are shown in Equation 9 and Equation 10 respectively. Answer option D is the correct answer option and option C is incorrect, but he believed them to both be correct. In the transcript below, Tim attempted a new technique, which I classify as *solving it by hand* to identify whether answer option C or D was correct. Unlike the presentation of the other substitution techniques described in this chapter I describe two hypothesized uses of this technique and discuss some of the uncertainty involved in classifying these hypothesized uses of the technique.

$$(\text{mult x y}) = (+\ \text{y}\ (\text{mult}\ (-\ \text{x}\ 1)\ \text{y}))$$

Equation 9. Correct recurrence relationship specified by answer option D.

Tim created the representation shown in Figure 55 and did so without tracing individual recursive calls. There are a number of aspects of his representation that are not explicit. He created this representation shown in Figure 55 during the following transcript.

> *"Well if we look at it this way. This one's going to be y plus (wrote "y+"), and assuming this works (pointing to answer option D) it's going to be x minus 1 times y. So it'll be like 4 y (wrote "4 y") so that'll be 5 y (wrote "= 5 y" on the second line), if we start with 5 y (wrote "5 y" on the first line). So like that should definitely work."*

5 y

y+ 4y = 5y

Figure 55 Inscriptions created by Tim to trace through answer option D

This technique can be used to evaluate the correctness of the recursive call as I described, but it is uncertain whether or not Tim used this technique. I interpret Tim's statements and inscriptions as indicating that he used the technique that I refer to as *solving it by hand*, but continue my narration of this case by discussing some of the assumptions in my interpretation of his solution.

Tim used the inscription of "5 y" in both the first and second lines of Figure 55 and I interpret the meaning of them differently. In the first line I interpret "5 y" as representing the initial function call to `mult` with the arguments 5 and y, which is typically written as `(mult 5 y)`. The "5 y" from the second line I interpret as representing the desired output of the function call `(mult 5 y)`. It is ambiguous if his inscription of "4 y" should be interpreted as mathematical notation for 4 times y or as shorthand for the recursive call `(mult 4 y)`. However, regardless of the interpretation of this inscription his statement "assuming this

www.manaraa.com

works" is consistent with the use of this technique and he did not show any indication of tracing a recursive call or referring to a previously calculated value.

At this point he became *"pretty confident"* that answer option D was correct. He said: *"So I'm thinking it is more likely to be this one, and I'm just not thinking this one (answer option C) through. I think it's D. I think it's D. I'm pretty confident."* Despite his confidence, he was still unable to use *simulating execution* to show that option C does not also produce the correct result. After two additional attempts to trace answer option C, I encouraged him to try to see if answer option C was correct using the technique he used when creating Figure 55. The recurrence relationship from answer option C is shown in Equation 10. Using this method he convinced himself that answer option C is incorrect.

$$(\text{mult x y}) = (\text{mult} (-\text{ x } 1) (+\text{ y y}))$$

Equation 10. Incorrect recurrence relationship specified by answer option C.

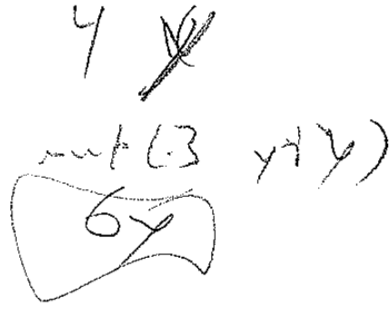


Figure 56 Inscriptions created by Tim to trace through answer option C

> *"Alright, well that reasoning is – that in theory multiply works. And does what we want. So if we start with 4 and um. 4 y (wrote 4y). When you run this, it's going to give us multiply (wrote "mult") three times y plus y (wrote "(3 y + y)"). And 3 times y plus y// 3 times 2 y is 6 y (wrote "6 y" and drew a box around it) So that's not right. And that would convince me I'm wrong (referring to his conclusion that answer option C was correct)."*

Like the first example where he proceeded by "assuming this works," here he explained his reasoning as "that in theory multiply works and does what you want." Using answer option C, Tim again traced a single execution of the recursive call specified by answer option C and substituted in the expected output of the `mult` function. His inscriptions during this are shown in Figure 56. Again he appeared to represent the initial function call of `(mult 4 y)` as "`4 y`." He was explicit about the recursive call to `mult` that would result. Applying the recurrence relationship from answer option C shown in Equation 10 and wrote "`mult (3 y+y)`."

Discussion

Techniques Beyond Substitution

The techniques identified in this chapter are not intended to be comprehensive of all possible techniques for reasoning about recursive functions. One technique, instead of using

www.manaraa.com

substitution, involves seeing that the algorithm described by a multiline summary of a function is the same as the algorithm known by the individual to perform the same calculation. This mapping allows an individual to conclude that the recursive function works as expected, but does not involve tracing specific values. This technique was used by only a single participant among 25 participants on the `mult` problem shown in Figure 44, which required the participants identify a recursive function that used repeated addition to multiply two numbers.

Example

When reasoning about the `mult` problem, the participant named Peter (participant identifier: Purple_TR) describes “x*y” as *“really saying x plus x plus x, y times.”* He observed that answer option D *“looks like it might do that”* and came to the correct conclusion that answer option D was the correct answer without ever tracing the function. During the segment documented in the following transcript, Peter created the representation shown in Figure 57[6].

> *“Umm, I can’t really explain it, but this seems reasonable as an answer. (Interviewer: OK, Why?) Umm, because I kind of think of multiplication as if we have x times y (wrote “x*y” shown in Figure 57) that’s really saying x+x+x, y times (completed inscriptions in Figure 57). So, and this looks like. Looks like it might do that, but I’m not sure (pause) So statement 1, umm. I guess that would imply this (points to answer option D), so I guess I’ll try D out first.”*

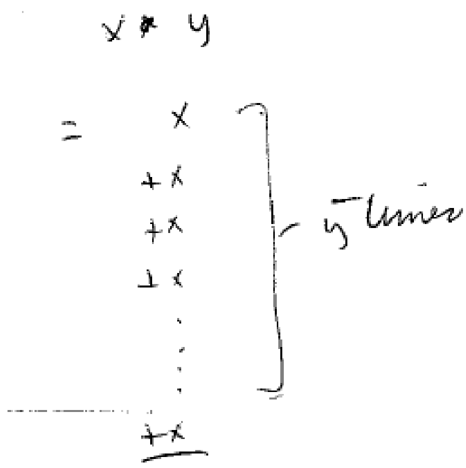


Figure 57. Peter's notes when explaining why answer option D was correct

After making the conclusion that D *“seems reasonable as an answer”* Peter used simulating execution to trace through each of the answer options with sample input. Although Peter did not use the technique to determine his final answer, this is a valid technique with which to reason about the function and does not involve substitution.

[6] The right most annotation may be read as “5 times,” but based upon Peter’s statements interpret it as “y times,” with the cross of the “t” extending above the “y.”

www.manaraa.com

Pedagogical Implications

The substitution techniques of *simulating execution*, *accumulating pending calculations*, and *memoization* all trace each recursive call, but do so in different orders. I hypothesize that the differences between these three techniques may provide the opportunity to scaffold students' understanding of how recursive functions are executed by computers and may provide the opportunity to highlight relevant features of the execution of recursive functions. For example, these substitution techniques and the corresponding representations for tracking state could be taught to students, which may support students in more accurately tracking state.

The substitution technique of *memoization* may be the most accessible to a novice student, because the student only needs to consider a single execution of the recursive function at a time. However, even without reasoning about an uninterrupted sequence of recursive calls, the student still has the opportunity to reason about the base case as producing a known value and the recursive expression producing a value that depends upon another execution of the recursive function.

The substitution technique of *accumulating pending calculations* also does not require reasoning about the execution of multiple recursive calls at a time, but provides the added complexity of considering each recursive call in an uninterrupted sequence. This uninterrupted sequence of recursive calls is identical to the sequence of recursive calls executed by a computer. Transitioning from the use of *memoization* to the use of *accumulating pending calculations* could potentially focus students on this sequence of recursive calls.

The substitution technique of *simulating execution* requires a more complete model of how a computer simulates execution of recursive functions. Connecting this technique to the substitution techniques of *accumulating pending calculations* may help students reason about the fact that the flow of control returns to the previous recursive calls. This feature of how a computer executes recursive calls is important for reasoning about non-linear recursion and recursion in an imperative programming environment, which both require returning to the previous recursive call to execute any remaining commands.

All of the substitution techniques build upon the algebraic technique of substitution. Legitimizing the use of this technique in the ways described above may be important to help students understand what of their content knowledge from math is applicable to solving computer science problems. The substitution technique of *solving it by hand* builds upon students' experience reasoning about algorithms, which may be another connection to students' out-of-domain knowledge. In the following section I describe how students' lack of knowledge about the legitimacy of using algebraic substitution in the ways described above may create a barrier to the transfer of this relevant skill. I hypothesize that this may be a more general pattern of difficulty when students are learning to apply elements of prior knowledge without explicit instruction legitimizing this transfer.

www.manaraa.com

Barriers to Transfer

In this chapter, I provided an example of how the participant named Emily used the substitution technique of *memoization* to solve the `WhatIsIt` problem shown in Figure 43. This example included statements from Emily discussing her concern about the legitimacy of this technique.

Emily was particularly articulate about her thought process and additional quotations regarding her lack of confidence with this technique are used to introduce the hypothesis that individuals' beliefs about the relevance of their prior knowledge from math or other domains may reduce the instances of productive transfer to the programming context.

Despite the fluidity of her use of *memoization* and the fact that at one point she described the process as *"obvious,"* she continually expressed hesitation about the legitimacy of the technique. For example, she expressed concern that she might be *"oversimplifying things"* and that she is *"not using the recursive calls properly."*

> *"That seems wrong to me. I feel like you're not taking it. Because it's supposed to go back to the case before. I think what I was just doing is thinking of it as math again, like you just, like when you solve two equations like if you do like, a system of equations, you just like take one equation and plug it in to the other one. I think that's what I'm doing here and I don't think I can just simply do that. I'm taking it like oh this is its own equation (pointing to WhatIsIt 4 2) so since, since it's like equivalent, 'oh it's 16' (with emphasis) I don't know if that makes sense but, but if it's it's own variable (points to what is it 4 2) like if you found out it was 16 before, oh you can just plug that in, so then it's like 16 times 4, but then, I'm not sure if it's how I can take, if it can still go back to the base case."*

In this case the technique Emily was using worked as intended. This is because the function she was tracing was written using a functional-programming paradigm, in which functions are guaranteed to produce the same output for a particular input and function calls have no side effects beyond producing an output.

Emily's concern about the legitimacy of this math-like technique is the foundation of my current hypothesis that individuals' beliefs about the relevance of their prior knowledge from math or other domains may reduce the instances of productive transfer to the programming context.

www.manaraa.com

CONCLUSION

Computer science education research has documented the pattern that many students are not successful learning to program (*e.g.*, McCracken *et al.*, 2001). However, it remains an open question what non-programming experiences could prepare students for success learning to program (Simon *et al.*, 2006). In this dissertation I investigate the question of what experiences students bring to the computer science classroom, how they can contribute to success, and how computer science pedagogy can take advantage of them. I hypothesize that students can make productive use of their out-of-domain knowledge and that this use may explain the range of novice students' success learning to program.

A common (Robins, 2010) alternative assumption among computer science educators and computer science education researchers is that innate aptitude for computer programming explains the range of students' success (Dehnadi, 2006; Lister *et al.*, 2004; Reges, 2008; Simon *et al.*, 2006). According to the work of Dweck (2007) and Steele (1997), when this assumption underlies pedagogy, student learning and attitudes suffer. This unproductive community assumption serves as motivation for this work, which identifies specific out-of-domain knowledge that may account for participants' successful reasoning. These examples of out-of-domain knowledge that support students' reasoning about computer science can contribute to dispelling the unproductive assumption that innate aptitude for computer programming explains the range of students' success.

To investigate what out-of-domain knowledge supports students' success, I conducted a detailed analysis of students' reasoning on computer programming questions that were identified by previous research as highly correlated with success on the AP CS exam (Reges, 2008). The participants were college students enrolled in one of three introductory programming courses at the University of California, Berkeley. As such, these students had been successful in their previous academic pursuits and could be expected to have a variety of out-of-domain knowledge, some of which may be relevant to reasoning about computer programming problems.

This dissertation is largely exploratory because little is known of how novice programmers build upon their out-of-domain knowledge. Unlike much of educational research that focuses only on students' persistent difficulties, in this dissertation I document competencies of novice programmers. These competencies are one set of results from the dissertation.

These competencies also motivated my development of hypotheses regarding the sources of these competencies, which was primarily theoretical work. To develop these hypotheses I applied learning theories many of which had not previously been applied to the domain of computer science.

These hypotheses varied in their level of speculation and additional research provides the opportunity for validation, refinement, or rejection. Below I provide a summary of the

www.manaraa.com

primary contributions from each analytic chapter and discuss the level of uncertainty in each chapter. I compare the uncertainties from each chapter for the purpose of calibration and while all analyses include uncertainty, I identify instances of more and less uncertainty within my analyses.

The coordination class of state

Summary

The first analysis chapter analyzes one student's moment-to-moment reasoning. The case shows an example where a student explicitly built upon everyday knowledge when constructing a scientifically normative explanation in the domain of computer science. In this case study I analyze the computer-science-specific and everyday knowledge of "and" that the participant used across four episodes. I use coordination class constructs to describe the ways in which the participant integrated her everyday knowledge of "and" into her reasoning about the computer science version of "and". This chapter contributes the first application of coordination class theory outside of physics and mathematics.

This student, Emily, reasoned about the behavior of the conditional "and," which from a computer science perspective requires reasoning about the input and output states of the conditional "and." I propose that state is a coordination class. To justify this claim I provide a description of the extent to which the concept of state meets the requirements for a coordination class.

Level of Uncertainty in Analysis

In episode two Emily attributed her knowledge as relevant for either "this world" or "the computer." Her statements about the nature of her knowledge provide me with a high level of confidence for the classification of some of her knowledge as out-of-domain knowledge. However, it remains an open question if this out-of-domain knowledge was primarily linguistic or had another source. Additional specificity regarding the nature of this knowledge would be beneficial to the goal of building upon students' out-of-domain knowledge.

The generalizability of Megan's success integrating her everyday knowledge of "and" in the computer science context is unknown and additional research could target these open questions and focus more narrowly on the hypothesized productivity of linguistic knowledge for reasoning about Boolean expressions in computer science.

Partial Descriptions of State Change

Summary

The second analysis chapter develops constructs to describe a type of knowledge that participants were believed to be using when reasoning about program state, which I refer to as *partial descriptions of state change.* This chapter emphasizes the nature of this knowledge and deemphasizes the dynamics of its use, which was the focus in the previous analysis chapter.

www.manaraa.com

In my future research the constructs that I develop in this chapter can be used for tracking the dynamics of participants' reasoning about state and I hypothesize that these partial descriptions of state change could serve as a concrete target for instruction.

Researchers have developed the hypothesis that the ability to produce a summary of code develops after the ability to trace code (Venables, Tan, & Lister, 2009). I observed the opposite pattern; participants generated summaries of code even when they were unsuccessful tracing the same code. This demonstrates that individuals' competence with tracing and summarizing code is context-specific; there may be no universal pattern of how these competencies develop and are used across contexts.

Level of Uncertainty in Analysis

The second analysis chapter sought to describe a particular competence rather than specify a source of knowledge. Labeling and describing a particular competence to develop a construct is also subjective, but the methods of evaluation are not the same as evaluating a potentially subjective analysis. Addition analysis, which applies these constructs, is necessary to determine the usefulness of understanding students' reasoning about computer program state.

Partial descriptions of state change may also be helpful for students to generate as preparation for tracing code as a mechanism for checking their individual steps. Independent of the empirical usefulness of these constructs, these constructs may be productive for communicating expectations to students' regarding desired summaries of code.

Intuitive knowledge about base cases and infinite loops

Summary

In the third analysis chapter I developed and present two hypotheses about what out-of-domain knowledge may have supported students' correct reasoning about infinite loops and base cases despite many of these students experiencing difficulty tracing the same function. I present a hypothesis that individuals' understanding of iterative processes may support their reasoning about infinite loops and that this knowledge of iterative processes could have the same properties as the type of intuitive knowledge that diSessa (1993) referred to as p-prims. I propose a new p-prim that includes this type of knowledge of iterative processes and refer to this as the *repeating p-prim.*

In the third analysis chapter I also present a second hypothesis that students' embodied experience may contribute to their reasoning about base cases in recursive functions. I document that participants used physical language when describing base cases and I developed two more specific hypotheses about the nature of this embodied knowledge. First, from examples of participants' physical language, I developed two metaphors that participants may have used in their descriptions of the base case in a recursive function. Second, participants' physical language also inspired my analysis that explains how the *blocking p-prim* (diSessa, 1993) can provide correct intuition about base cases in recursive functions.

www.manaraa.com

In developing the hypotheses presented in analysis chapter three I connect both p-prim theory and Metaphor Theory (Lakoff & Núñez, 2000) with computer science education. In this application of p-prim theory, I propose a clarification to p-prim theory, which is that although p-prims provide the expectation that a phenomenon does not need an explanation, if it is brought to the individual's attention he or she may still be able to reason articulately about the need for an explanation.

Level of Uncertainty in Analysis

The third analysis chapter was the most speculative. I applied both p-prim theory (diSessa, 1993) and Metaphor Theory (Lakoff & Núñez, 2000), which have not previously been applied to computer science. This analysis was a first step toward identifying the source and content of students' relevant out-of-domain knowledge and the hypotheses that students built upon the repeating and blocking p-prims may be validated, refined, or rejected by additional research.

Substitution techniques

Summary

The final analysis chapter documents participants' application of algebraic substitution techniques to the task of tracking program state in recursive functions, which is an additional example of how participants used out-of-domain knowledge when solving computer science problems. The content of this chapter is intended to be valuable to computer science educators and it describes what appear to be four distinct instantiations of algebraic substitution to track program state. The fourth analysis chapter proposes a progression of substitution techniques to scaffold students to reason about the execution order of recursive functions. The chapter functions as a first step toward a taxonomy of how algebraic substitution techniques can be applied to tracing the state of recursive functions.

Level of Uncertainty in Analysis

The fourth analysis chapter was the most specific in identifying the likely source of knowledge for tracing program state as from experience with algebraic substitution. In the analysis I claim that the techniques can be seen as applications of algebraic substitution. Emily identified this connection, but I do not make the claim that other students made or would make this connection. If students generally rejected this connection between algebraic substitution and recursion, the connection would be unlikely to be pedagogically valuable.

Summary of Contributions

This dissertation applied learning theories that had not previously been applied to computer science education. Through this application I extend the learning theories to the domain of computer science, propose refinements to the theories, and provide insights into participants' reasoning about particular computer science topics. While open questions remain, this dissertation provides first steps toward identifying out-of-domain knowledge that students can apply to solving computer science problems.

www.manaraa.com

REFERENCES

Aronson, J., Lustina, M. J. Good, C., Deough, K., Steele, C. M., & Brown, J. (1999). When white men can't do math: Necessary and sufficient factors in stereotype threat. Journal of Experimental Social Psychology 35 (29), 29-46.

Barker, L. J., McDowell, C., & Kalahar, K. (2009). Exploring factors that influence computer science introductory course students to persist in the major. Proceedings of the 40th SIGCSE Technical Symposium on Computer Science Education, Chattanooga, TN, 282-286.

Ben-Ari, M. (2001). Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1), 45–73.

Biggs, J. B. (1999): Teaching for quality learning at University, Buckingham. Open University Press.

Biggs, J. B. & Collis, K. F. (1982): Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). New York, Academic Press.

Carr, P. B. & Steel, C. M. (2009). Stereotype threat and inflexible perseverance in problem solving. Journal of Experimental Social Psychology. 45, 853-859.

Clancy, M. (2004). Misconceptions and Attitudes that Interfere with Learning to Program. In Fincher, S. & Petre, M. (Eds.), *Computer Science Education Research* (pp. 85-100). New York: Taylor & Francis.

Cobb, P.; Confrey, J.; DiSessa, A.; Lehrer, R.; Schauble, L. (2003), "Design experiments in educational research", Educational Researcher 32 (1): 9–13,

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *The journal of computing in small colleges*. ACM. 107-116.

Corbin, J. M., & Strauss, A. C. (2008). Basics of Qualitative Research. Thousand Oaks, CA: SAGE Publications.

Dehnadi, S. (2006). Testing Programming Apptitude. In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). *Proc. PPIG 18 (Brighton, UK, September 07 – 08, 2006). PIGG '06*. 22-37.

diSessa, A. A. (1986). Models of Computation, in User Centered System Design: New perspectives on Human-Computer Interaction, (Eds. D. A. Norman and S. W. Draper) Hilsdale, New Jersey: Lawrence Erlbaum Associates Inc., 201-218.

diSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and Instruction*, *10* (2-3), 105-225.

diSessa, A. A., (2000). *Changing Minds: Computers, learning and literacy*, MIT Press.

diSessa, A. A. (2006). A history of conceptual change research: threads and faultlines. In K. Sawyer (Ed.), Cambridge handbook of the learning sciences. Cambridge, UK: Cambridge University Press.

diSessa, A. A. (2007). An interactional analysis of clinical interviewing. *Cognition and Instruction.* 25*(4)*, 523-565.

www.manaraa.com

diSessa, A. A., & Minstrell, J. (1998). Cultivating conceptual change with benchmark lessons. In J. G. Greeno & S. V. Goldman (Eds.), *Thinking Practices in Mathematics & Science Learning.* Mahwah, NJ: Lawrence Erlbaum Associates, 155-187.

diSessa, A. A., & Sherin, B. L. (1998). What changes in conceptual change? *International Journal of Science Education*, *20*(10), 1135-1191.

diSessa, A. A., & Wagner, J. F. (2005). What coordination has to say about transfer. In J. Mestre (ed.), *Transfer of learning from a modern multi-disciplinary perspective* (pp. 121-154). Greenwich, CT: Information Age Publishing.

du Boulay, B., O'Shea, T. & Monk, J. (1989). The black box inside the glass box: Presenting computing concepts to novices, in Studying the Novice Programmer (E. Soloway & J.C. Spohrer, Eds.) Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc., 431-446.

du Boulay, B. (1989). Some difficulties learning to program, in Studying the Novice Programmer (E. Soloway & J.C. Spohrer, Eds.). Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc., 283-299.

Dweck (2007). Mindset: The new psychology of success. New York, NY: Random House, Inc.

Dweck, C. S. & Leggett, E. L. (1988). A social-cognitive approach to motivation and personality. *Psychological Review.* 95(2), 256-273.

Engle, R. A., Conant, F. R. & Greeno, J. G. (2007). Progressive refinement of hypotheses in video-supported research. In R. Goldman, R. Pea, B. J. Barron & S. Derry (Eds.), *Video research in the learning sciences* (pp. 239-254). Mahwah, NJ: Erlbaum.

Evans, D. W. (1983). Understanding zero and infinity in the early school years (Unpublished doctoral dissertation).University of Pennsylvania, Philadelphia.

Falk R. (2010). The Infinite Challenge: Levels of Conceiving the Endlessness of Numbers. *Cognition and Instruction*, 28(1), 1-38.

Fleury, A. (1993). Student beliefs about Pascal programming. *Journal of Educational Computing Research*, 9(3), 355-371.

Friedman, D. P. & Felleisen, M. (1996). The Little Schemer - 4th Edition. MIT Press.

Garcia, D. D., Harvey, B., & Segars, L. (2012). CS pirnicples pilot at University of California, Berkeley. ACM Inroads. 58-60.

Garvin-Doxas, K. & Barker, L. J. (2004). Communication in computer science classrooms: Understanding defensive climates as a means of creating supportive behaviors. *Journal of Educational Research in Computing*, *4(1)*, 1-18.

George, C. E. (2000) Experiences with Novices: The Importance of Graphical Representation in Supporting Mental Models. In A. F. Blackwell & E. Bilotta (Eds). Proc. PPIG 12

Goff, P. A. Steele, C. M. & Davies, P. G. (2008) The space between us: Stereotype threat and distance in interracial contexts. *Journal of Personality and Social Psychology. 94(1)*, 91-107.

www.manaraa.com

Kahney, H. (1989) What do novice programmers know about recursion? *Studying the Novice Programmer* (E. Soloway & J.C. Spohrer, Eds.) Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc., 209-228.

Kurland, D. M., & Pea, R. D., (1989). Children's mental models of recursive logo programs. *Studying the Novice Programmer (E. Soloway & J.C. Spohrer, Eds.)* Hillsdale, New Jersey: Lawrence Erlbaum Associates Inc., 315-323.

Hammer, D. (2000). Student resources for learning introductory physics. American Journal of Physics. 68(1), 52-59.

Hammer, D., Elby, A., Scherr, R. E., & Redish, E. F. (2004). Resources, framing, and transfer. In J. Mestre (Ed.), Transfer of Learning: Research and Perspectives

Harvey, B., & Mönig, J. (2010). Bringing 'No Ceiling' to Scratch: Can one language serve kids and computer scientists? *Constructionism*, 2010, 1–10.

Hoadley, C. M., Linn, M. C., Mann, L. M., & Clancy, M. J. (YYYY) When, why and how do novice programmers reuse code? *Proceedings of the Sixth Workshop on Empirical Studies of Programmers,* W. D. Gray & D. A. Boehm-Davis (Eds.), Ablex Publishing, 1996.

Lakoff, G., & Núñez , R. E. (2000). Where mathematics comes from: How the embodied mind brings mathematics into being. New York, NY: Basic Books.

Lakoff, G. and Johnson, M. (1980). *Metaphors we live by.* Chicago: University of Chicago Press.

Leron, U. & Zazkis, R. (1986). Computational Recursion and Mathematical Induction. *For the Learning of Mathematics, 6(2)*. 25–28

Levrini, O. & diSessa, A. A. (2008) How students learn from multiple contexts and definitions: Proper time as a coordination class. *Physics Education Research,* 4, 010107.

Lewis, C., (2007). Attitudes and Beliefs About Computer Science Among Students and Faculty. *ACM SIGCSE Bulletin 39(2)*, 37-41.

Lewis, C. M., Yasuhara, K., & Anderson, R. E. (2011). Deciding to Major in Computer Science: A Grounded Theory of Students' Self-Assessment of Ability. Proceedings of the International Computer Science Education Research Workshop. Providence, RI. 3-10.

Lewis, C. M., Titterton, N., & Clancy, M. (2012). Using Collaboration to Overcome Disparities in Java Experience. Proceedings of the International Computer Science Education Research Workshop. Auckland New Zealand.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Mostrom, J.E., Sanders, K., Seppala, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. Working group reports from ITiCSE on Innovation and technology in computer science education,119-150.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008) Relationship between reading, tracing and writing skills in introductory programming. *Proceedings of the fourth International Workshop on Computing Education Research.* 101-111.

www.manaraa.com

Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. (2008). Programming by Choice: Urban Youth Learning Programming with Scratch. ACM Special Interest Group on Computer Science Education., Portland: ACM.

Margolis, M., Estrella, R., Goode, J., Jellison Holme, J., & Nao, K. (2008). Stuck in the Shallow End: Education, Race, and Computing (MIT Press)

McCracken, W.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. *ACM SIGCSE Bulletin 33(4):125-140.*

Monaghan, J. (2001). Young people's ideas of infinity. *Educational Studies in Mathematics, 48, 239–257.*

NCWIT (2009). By The Numbers. Retrieved from http://www.ncwit.org/pdf/BytheNumbers09.pdf

Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books, Inc.

Parnafes, O. (2007) What does "fast" mean? Understanding the physical world through computational representations. The Journal of the Learning Sciences. 16(3), 415-450.

Pennington, N. (1987). Comprehension strategies in programming. In G.M. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop (pp. 100 – 113). Norwood, NJ: Ablex Publishing Company.

Philpott, A, Robbins, P., and Whalley, J. (2007): Accessing the Steps on the Road to Relational Thinking. 20th Annual Conference of the National Advisory Committee on Computing Qualifications, Nelson, New Zealand, 286.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass; teaching CS0 with Alice. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, Covington, KY, 213-217.

Reges, S. (2008) They mystery of b := (b = false). *ACM SIGCSE*, 39, 21-25.

Robins, A. (2010). Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, 20(1), 37-71.

Russ, R. S. & Sherin, B. (2008). Reframing research on intuitive science knowledge. *International Conference on Learning Sciences*. 2, 279-286.

Sajaniemi, J. (2002) Visualizing Roles of Variables to Novice Programmers. In J. Kuljis, L. Baldwin & R. Scoble (Eds). *Proc. PPIG 14.* 111-127.

Sajaniemi, J. & Kuittinen, M. (2005) An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education*, 15(1), 59 - 82.

Sajaniemi, J., Kuittinen, M., & Tikansalo, T. (2008) A study of the development of students' visualizations of program state during an elementary object-oriented programming course. Journal on Educational Resources in Computing. 7(4) 1-31.

Schiralli, M. & Sinclair, N. (2003). A constructive response to 'Where mathematics comes from' *Educational Studies in Mathematics*. 52(1). 79-91.

www.manaraa.com

Shinners-Kennedy, D. (2008). The everydayness of threshold concepts: `State' as an example from computer science. In Land, R., Meyer, J. H., and Smith, J., (Eds.), *Threshold Concepts Within the Disciplines* (119-128). Sense Publishers.

Simon, Cutts, Q., Fincher, S., Haden, P., Robins, A., Sutton, K., Baker, B., Box, I., de Raadt, M., Hamer, J., Hamilton, M., Lister, R., Petre, M., Tolhurst, D., Tutty, J. (2006). The ability to articulate strategy as a predictor of programming skill. *Proc Eighth Australasian Computing Education Conference, Hobart, Australia*, Jan 2006.

Simon, B., Hanks, B., Murphy, L., Fitzgerald, S., McCauley, R., Thomas, L., & Zander, C., (2008b). Saying Isn't Necessarily Believing: Influencing Self-theories in Computing. *ICER*, 173-184.

Smith, J. P., diSessa, A. A., & Roschelle, J. (1993). Misconceptions reconceived: A Constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*, ***3***(2), 115-163.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive Strategies and Looping constructs: An Empirical Study. *Communications of the ACM, 26(11)*, 853-860.

Spencer, S. J., Quinn, D., C. M. Steele (1999). Stereotype threat and women's math performance. *Journal of Experimental Psychology*, 35, 4-28.

Steele, C. M. (1997). A threat in the air: How stereotypes shape intellectual identity and performance. *American Psychologist, 52(6)*, 613-629.

Steele, C. M., & Aronson, J. (1995). Stereotype threat and the intellectual test performance of African Americans. *Journal of Personality and Social Psychology. 69(5)*, 797-811.

Strike, K.A. & Posner, G.J. (1992) A Revisionist Theory of Conceptual Change. *In P.A. Duscal & R.J. Hamilton (Eds.) Philosophy of Science, Cognitive Psychology, and Educational Theory & Practice.* Albany, NY: SUNY Press.

Thaden-Koch, T. C., Dufresne, R. J., & Mestre, J. P. (2006). Coordination of knowledge in judging animated motion. *Physical Review Special Topics – Physics Education Research, 2(2)*, 1-11.

Titterton, N., Lewis, C. M., & Clancy, M. (2010). Experiences with lab-centric instruction. *Computer Science Education (Ed. Y. Ben-David Kolikant) 20(2)*, 79-102.

Venables, A., Tan, G., & Lister, R., (2009). A closer look at tracing, explaining code writing skills in the novice programmer. Proceedings of the fifth International Workshop on Computing Education Research. 117- 128.

Vosniadou, S., & Brewer, W. F. (1992). Mental models of the earth: A study of conceptual change in childhood. *Cognitive Psychology*, *24*, 535-585.

Wagner, J. F. (2006). Transfer in Pieces. Cognition and Instruction, 24(1), 1-71.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., & Prasad, C. (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. 8th Australasian Computing Education Conference, Hobart, Australia. 243-252.

Wittman, M. C. (2001). The object coordination class applied to wavepulses: Analysing student reasoning in wave physics. International Journal of Science Education.

www.manaraa.com