EMBODIMENT IN COMPUTER SCIENCE LEARNING: HOW SPACE, METAPHOR, GESTURE, AND SKETCHING SUPPORT STUDENT LEARNING

A Dissertation
Presented to
The Academic Faculty

By

Amber Solomon

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology

May 2021

© Amber Solomon 2021

EMBODIMENT IN COMPUTER SCIENCE LEARNING: HOW SPACE, METAPHOR, GESTURE, AND SKETCHING SUPPORT STUDENT LEARNING

Thesis committee:

Dr. Betsy DiSalvo, Co-Advisor
School of Interactive Computing
Georgia Institute of Technology

Dr. Mark Guzdial, Co-Advisor
Electrical Engineering and Computer Science
University of Michigan

Dr. Ashok Goel
School of Interactive Computing
Georgia Institute of Technology

Dr. Wendy Newstetter
School of Interactive Computing
Georgia Institute of Technology

Dr. Ben Shapiro
College of Education and Human Development
Georgia State University

Dr. David Uttal
School of Education and Social Policy
Northwestern University

Date approved: April 12, 2021

ACKNOWLEDGMENTS

Too many to thank... TBD

TABLE OF CONTENTS

Acknowledgments iii

List of Tables x

List of Figures xi

Summary xiv

Chapter 1: Introduction 1

1.1 Research Motivation 5

1.2 Research Goals and Questions 7

1.2.1 Research Agenda 8

1.3 Dissertation Overview 10

1.4 Positionality 11

Chapter 2: Setting the Context of Recursion 13

2.1 Recursion in Programming—Recursive Invocation 14

2.2 Recursive Execution 15

2.3 Learning Recursion 20

2.4 Mental Models of Recursion 21

Chapter 3: Conceptual Framework 25

3.1 Embodiment 25

3.2 Metaphor 27

3.2.1 Metaphors support Understanding and Reasoning 29

3.2.2 Metaphorical Construals 31

3.3 Gestures 32

3.3.1 Gestures Defined 33

3.3.2 Gesture Can Reveal What a Learner Knows 33

3.3.3 Gesture Production Can Support Learning 34

3.3.4 Seeing Gesture Can Support Learning 34

3.3.5 The Connection Between Gesture and Learning 34

3.3.6 Gesture Taxonomy 35

3.4 Tool Use: Sketching 35

Chapter 4: Related Work 39

4.1 Complexities of Learning to Program 39

4.2 Embodiment and CS Learning 41

4.3 Metaphors 43

4.4 Gesture 46

4.4.1 A Gesture Taxonomy in Computing Education 48

4.4.2 An Exploratory Observational Study 48

4.4.3 The Types of Gesture in a Computing Classroom 50

4.4.4 Discussion 57

4.4.5 Conclusion 58

4.5 Sketching 59

Chapter 5: Embodied Representations in Computing Education: How Gesture, Metaphor, and Sketching Support Teaching Recursion 61

5.1 Introduction 61

5.1.1 Motivation 62

5.2 Background 63

5.3 Method 64

5.3.1 Data Sources 64

5.3.2 Data Analysis 65

5.4 Case Studies 66

5.4.1 Case Study 1: Gesture 66

5.4.2 Case Study 2: Metaphors and Sketching 68

5.5 Discussion and Conclusion 70

Chapter 6: ”On the Reality of Teaching Programming”: Interpreting Embodiment in CS Classrooms 75

6.1 Introduction 75

6.2 Multimodality and Communicating Information 78

6.3 Methods 80

6.3.1 Data Collection 81

6.3.2 Data Analysis 82

6.4 Reflexivity 87

6.5 Findings 87

6.5.1 Gesture Production 88

6.5.2 Metaphors 90

6.6 Case Studies 94

6.7 Discussion 106

6.7.1 Pedagogical Function 107

6.7.2 Challenges with Interpretation 108

6.7.3 Designing Metaphors 110

Chapter 7: How Students Use Conceptual Blends, Metaphors, and Embodiment to Make Sense of Computation 112

7.1 Introduction 112

7.2 Conceptual Blending 113

7.3 Methods 116

7.3.1 Participant Recruitment and Compensation 117

7.3.2 The Course 118

7.3.3 Participants 119

7.3.4 Data Collection 120

7.3.5 Dyad Protocol 121

7.3.6 Interview Questions 122

7.4 Data Analysis 123

7.4.1 Case Selection 123

7.4.2 Coding Gestures 124

7.4.3 Metaphor 125

7.4.4 Parsons problems 126

7.4.5 Sketches 126

7.4.6 Segmentation . . . 127
7.4.7 Tradeoffs and Limitations . . . 128
7.5 Findings . . . 130
7.5.1 Overview of the Problems . . . 130
7.5.2 Gestures . . . 137
7.6 Case Studies . . . 141
7.6.1 Case Study 1 . . . 141
7.6.2 Case Study 2 . . . 146
7.6.3 Case Study 3 . . . 152
7.7 Discussion and Implications . . . 156
7.7.1 Sense-Making Resources . . . 156
7.7.2 Sketching and Gesture . . . 157
7.7.3 Learning Implications . . . 158

Chapter 8: Conclusion and Implications . . . 160
8.1 Implications . . . 162
8.1.1 Learning Implications . . . 162
8.1.2 Conceptual Framework . . . 163
8.1.3 Pedagogical Implications . . . 164
8.2 Contributions . . . 165
8.3 Future Work . . . 166

Appendices . . . 168
Appendix A: Demographic Survey . . . 169

Appendix B: Dyad Student Sketches 177

Appendix C: Code Trace for Each Problem 182

References 184

LIST OF TABLES

2.1 Most commonly used conceptual models. Table adapted from [50]. 17

2.2 Mental models of recursion. Adapted from [50] 22

5.1 A summary of the embodied representations teachers used. 71

6.1 Professors' demographics and video information 83

6.2 List of Some Metaphors 93

7.1 Moves and Descriptions 127

7.2 Different kinds of metaphorical construals. 140

LIST OF FIGURES

2.1 The code for the function factorial. 15

2.2 Non-tail end execution 16

2.3 A representation of a stack in CS. The first image shows elements pushing into the stack. The second image shows elements popping out of the stack. . 17

2.4 A representation of a flow chart of the factorial function 18

2.5 A representation of a code trace of the factorial function 19

2.6 Copies and looping mental models of the factorial function [55] 23

4.1 Code for a non-tail end recursive function. 41

4.2 A student using deictic gestures while they trace a program's control flow. . 52

4.3 A student using a metaphoric gesture while they described a loop. 56

5.1 The professor uses metaphoric and deictic gesture to describe a list. At the top of this figure is a typical conceptualization of a list. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image. 67

5.2 The professor uses metaphoric gesture while describing the tree and children nodes. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image. 68

5.3 The professor uses metaphoric gesture while describing the tree and children nodes. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image. 69

5.4 The professor sketching a code trace while using embodied language. The professor's talk indicating embodied language are noted: stepping out±, diving in¢, ontological metaphor° 71

6.1 The approach to determine if a word or utterance is a metaphor [160]. . . . 86

6.2 Teachers' gestures while describing memory and the call stack. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image. . . . 89

6.3 Professor making a cyclical gesture. 95

6.4 Professor uses iconic gesture while describing searching through a phone book. 97

6.5 The professors code, with the three lines the student refers to in a white box. 103

7.1 Diagram of a Generic Conceptual Blend. There are two input spaces that are "blended" into the third, blended space. 116

7.2 Problem 1 131

7.3 Problem 2 133

7.4 The solution to problem 2 is adding the line in the square. 134

7.5 Problem 3 135

7.6 The solution to Problem 3. 136

7.7 Lee uses noniconic gesture to describe recursion. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image. 139

7.8 Aru's blend to predict when the recursive process ends. Note, the parts in blue is the base case blend. 143

7.9 Diagram of exponents blend. 145

7.10 Aru's stacking gesture. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image. 146

7.11 Diagram of when the base case should terminate. The diagram in blue is the blend about variables. 149

7.12 Diagram of when to put lines of code blend. 151

7.13 Diagram of "to the left" blend. 153

7.14 Diagram of "get N back" blend. 155

C.1 Code trace for problem 1. 182

C.2 Code trace for problem 2. 183

C.3 Code trace for problem 3. 183

SUMMARY

Several recent studies in computing education research (CER) have found spatial thinking may be influential in learning computer science (CS). Correlational studies have demonstrated that psychometrically assessed spatial skills predict performance in introductory CS courses [1, 2, 3]. However, it's hard to explain these results. There is not an obvious match between the logic for computer programming and the logic for thinking spatially. CS is not imagistic or visual in the same way as other STEM disciplines, since students can't see bits or loops. Margulieux suggests that spatial skill is a predictor of success in some STEM disciplines because problem solving in those disciplines requires spatial reasoning about some object [4]. For example, Kozhevnikov and colleagues found that in physics, spatial ability is related to solving kinematics problems, particularly predicting the motion of an object [5].

The goal of this dissertation is to document how space influences appears in CS learning. Embodiment is a theoretical framework which I use to document the ways space and embodied representations (i.e., gesture, sketching, and metaphor) appear while students make sense of computation and express computational ideas. The implication is that people make meaning from their body-based, lived experiences, not just through their minds [6], even for computing in its virtual nature.

However, the contributions of the body in sense-making is understudied in CER. Some research has suggested that embodiment is simply pedagogical; it provides a way to explain computing concepts [7, 8]. Other research has taken a disembodied view of CS learning. Dijkstra was antagonistic towards using metaphor and body-based experiences to teach and learn CS. He argued that attempting to link what was familiar to computation is ”hopelessly inadequate” [9]. More recently, Cao and collaborators conducted an experiment to determine if analogies were ”valuable” for CS learning. They conclude that analogies are not valuable for long term learning or knowledge transfer and present inconclusive evidence

that analogies are valuable for short term learning [10].

In this dissertation, I describe three studies. In the first study, I conducted a grounded analysis of a set of naturalistic video recordings of university professors teaching recursion in their classrooms. I wanted to understand how teachers use space in CS classrooms. Therefore, I looked for spatial representations, including iconic gestures, spatial language and spatial metaphors, and artifacts, such as sketches or diagrams. While conducting this analysis, I noticed an interesting relationship between space and embodiment. The gestures the teachers created about computation seemed to be enacted metaphors; that is, the gestures seemed to act out a metaphor. Similarly, the teachers used spatial language (e.g., then, it goes down here; then, I jump up to the if-statement) while describing a code trace and used interesting language, such as calls, runs, and bound, to refer to function invocation, program execution, and variable assignment, respectively. However, all these are metaphors. The computer is not going down or calling anything, but teachers used metaphors and physicality to represent the abstract. Moreover, teachers metaphorically construe themselves as executing lines of code or constructing an agent who does that. Embodiment seems to play a central role in the ways teachers communicate information and meaning. This study sets forth the conceptual framework of the kinds of embodiment used to analyze the rest of the work.

In the second study, I considered learning by reflecting on how teachers communicate and structure learning opportunities—as a way to understand what students need to interpret in order to learn. Using the same data corpus from the first study, I sought to understand how well students could use or interpret the representations the teachers created and communicated. I focused this examination on gestures and metaphors, as only one teacher created a sketch. In my entire data corpus, embodiment was the only representation that the teachers used to make computation, physical. I found that while embodiment could have pedagogical significance, the ways teachers use it could be a source of confusion and add to the list of difficulties that make learning to program hard. Teachers randomly switch

perspectives, refer to lines of code as if they exist in the real world, and create gestures that are either communicating important information or could also be nonsense. Professors use metaphors, such as calls and runs, without explicitly stating what they mean. The embodiment, specifically metaphors, do not seem to be intentionally designed for explanatory power.

In the final study, I video recorded 10 dyads of college students while they solve recursive problems. While conducting another grounded analysis, I found that the students creatively constructed different metaphors and overlaid metaphors to make sense of computation. I used conceptual blending as a theoretical framework to describe the ways in which they blended multiple metaphors that allowed them to make sense of computation, to reason about the behavior of code and make predictions or explanations. I conclude by making a case that it is not about designing an everlasting representation that always helps one understand this concept. It is about finding a cognitive toehold on understanding this problem right now.

CHAPTER 1
INTRODUCTION

As humans, we live in a physical world. We live in a space. We often think about space in ways that we might not realize. Spatial reasoning has been found to be influential in learning algebra, biology, and other science, technology, engineering, and math (STEM) subjects [11]. Recently, studies have found that spatial ability may be influential in learning computer science (CS).

Correlational studies have demonstrated that psychometrically assessed spatial ability predict performance in introductory CS courses [1, 2, 3]. Cooper et al. found a correlation between students' spatial ability scores and their ability to learn to program [1]. They also found that when students with low socioeconomic status (SES) were given spatial skills training, it improved their CS knowledge just as much as higher-SES students did [1]. Recently, Bockmon et al. replicated and extended Cooper et al.'s study and found that students' performance on spatial ability assessments and a CS content instrument improved when they received spatial skills training [12].

Inspired by Cooper et al.'s study [1], my colleagues and I sought to understand the variables that might mediate or explain why low-SES students typically perform worse in CS classes. We found that scores on spatial ability tests were a better mediating variable to explain CS performance than students' past access to computing experiences was [2].

However, explaining these results is difficult. As Uttal and Cohen said, correlation is not causation, and these correlations could be due to several reasons unrelated to spatial skills [13]. Nonetheless, the results are intriguing when put into conversation about the potential implications of how students make sense of computation and the types of reasoning they might use to do so. If we were confident about the relationship, we could teach CS differently, which might improve success and retention in CS.

However, it’s hard to explain these results. There is not an obvious match between the logic for computer programming and the logic for thinking spatially. CS is not imagistic or visual in the same way as other STEM disciplines, since students can’t see bits or loops. Spatial abilities and STEM performance are highly correlated, but that makes sense because STEM is a highly visual space. CS is not inherently visual; it is a science of the artificial. This means that CS is something that was designed and not inherited from nature [14].

Margulieux suggested that spatial ability is a predictor of success in some STEM disciplines because problem solving in those disciplines requires spatial reasoning about some object [4]. For example, Kozhevnikov and colleagues found that in physics, spatial ability is related to solving kinematics problems, particularly predicting the motion of an object [5].

I define spatial ability as the ability to use space to both ”think with” and ”think through.” Therefore, spatial ability is a problem-solving activity, helping one reason about and make sense of phenomena. It also serves communicative functions. It gives us a language or way to talk about phenomena and gives us a way to jointly make meaning and collaborate.

To think more deeply about this problem, in this dissertation, I documented how space appears in CS learning. My dissertation is naturalistic and inductive, as little is known about how space influences CS learning. Embodiment is a theoretical framework that I use to document the ways in which space and embodied representations (i.e., gestures, sketching, and metaphors) appear while students make sense of computation and express computational ideas. The implication is that people make meaning from their body-based, lived experiences and not just through their minds [6], even in a discipline such as computing, which is virtual in nature.

While looking for space, I recognized the importance of embodiment and metaphor, which was different than the CEd literature might expect. For example, teachers use the following spatial language when describing a code trace: then, it goes up here before going

back down to the if-statement. The code is not actually going anywhere, but physicality and embodiment are used to explain the abstract concept.

Embodiment as a theoretical framework offers two related novel ways to investigate the relationship between space and computation. First, embodiment encourages investigations into how students use their bodies and body-based experiences to make the digital physical, which has implications for understanding how and through what mechanisms students make sense of computation. Research in STEM education has found that students often externalize or offload their spatial thinking using embodied representations; this embodiment offers students a way in to manage and think with the abstract [15]. For example, Enyedy and colleagues documented a case of a student slipping on linoleum, which later helped them reason about how the speed of an object increases [16]. Similarly, Roschelle describes students using metaphors, like pulling, to reason about acceleration [17].

However, the contributions of the body to sense-making are understudied in computing education research (CER). Some research has suggested that embodiment is simply pedagogical; it provides a way to explain computing concepts [7, 8]. Other studies have taken a disembodied view of CS learning. Dijkstra was notably antagonistic toward the use of metaphors and body-based experiences to teach and learn CS. He argued that attempting to link what was familiar to computation is “hopelessly inadequate” [9]. More recently, Cao and collaborators conducted an experiment to determine whether analogies were valuable for CS learning. They concluded that analogies are not valuable for long-term learning or knowledge transfer, and they presented inconclusive evidence that analogies are valuable for short-term learning [10].

Second, using embodiment moves the focus of analysis outside the head. Moving outside the head allows us to interrogate the practices and sociocultural contexts that help students with understanding and reasoning. Interrogating practices has direct implications for pedagogy; this process gives us the opportunity to explore how we might design learning environments that better leverage students’ lived experiences.

Moreover, moving outside the head gives us different ways to think about the relationship between space and computation. Situative theories suggest that spatial abilities involve the ability to use space in order to solve problems and are mediated by the social interactions and tools afforded within a specific context [18]. Perhaps students use their understanding of space to comprehend or imagine constructs in CS.

However, CER has only looked inside the head to understand the relationship between space and computation. Researchers have proposed models that explain the relationship between spatial skills and learning CS at the cognitive level [3, 4]. Marguiliuex proposed the spatial encoding strategy theory, which focuses on the ability to create and represent mental encodings of information [4]. Looking inside the head is only a partial and contingent understanding of the relationship between spatial skills and learning CS [19]. We need many avenues of investigation if we want a complete understanding of how spatial skills operate in learning CS.

I investigated learning through two avenues. The first is as a sense-making, problem-solving activity, and the second is as a meaning-making and social process between teachers and students. In some ways, I was inspired to understand what was actually happening in these classrooms and how students are actually learning and what mediates or supports that learning.

This dissertation comprises three studies, each providing insight into how students or teachers use embodiment to make sense of computation or communicate computation. The first used grounded theory to analyze a video corpus of CS professors teaching recursion to their class. From this study, I developed a conceptual framework of the different kinds of embodiment teachers used. The second study used the same video corpus as the first study and hypothesized about how well students can interpret the teacher's embodiment to support their learning. The final study described the ways in which students use embodiment and conceptual blends to make sense of computation. It analyzed pairs of students as they solved recursive problems.

To consider a discrete problem space, I study recursion. Recursion is one of the few topics in CER that has been studied intensively compared with other programming constructs and concepts and understanding recursion conceptually and, to an extent, implementing a recursive call require an understanding of different CS concepts, including function invocation, return statements, stacks, and memory allocation,

1.1 Research Motivation

My research uses qualitative methods to investigate how embodiment appears and influences CS learning. To study embodiment, I will examine how embodied representations (i.e., gestures, sketching, and metaphors) support CS learning. These representations were the most salient in computing classrooms and in students' problem solving. This qualitative approach was inspired by two different perspectives.

The first perspective was inspired by a statement made by computing education researcher Ben Shapiro. Shapiro argued that curriculum, programming languages, representations, assessments, and so on are all designed, socially negotiated, and situated [20]. Shapiro's argument was that we cannot understand CS learning without comprehending these designed, sociocultural, and sociopolitical contexts. This, in some ways, turns many CS learning problems and solutions into design challenges.

CS is not inherently visible; we do not inherit it from nature. Studies on students' and teachers' usage of representations in CS classrooms agree that visual representations are important in CS education because they create something tractable for students to develop conceptual understandings with [21, 22, 23].

I became interested in representations with two past projects that sought to make CS visible. The first was a project in which we used studio pedagogy and augmented reality (AR) to make CS visible [24]. We wanted to know whether it is possible to create a non-defensive CS classroom environment by making students' work visible. We found that the CS classroom culture can be reoriented from having a defensive climate to adopting a

culture that supports a community of learners when students' work is made visible. We hypothesize that by making students' work visible (literally on the walls of an AR-enabled design studio), we allow students to use their spatial and social skills in order to engage differently with learning CS.

My second interest (described in more detail in Chapter 4) came from a study in which I developed a conceptual framework to support future learning and teaching studies that incorporate gesture studies in programming contexts [25]. I observed a high school CS class and matched the gestures that students created to an existing taxonomy of gestures from mathematics education. In particular, I introduced how gesture has been used to study teaching and learning, with a focus on one discipline (mathematics), as well as critically reviewed and interpreted the concepts that may be most relevant to programming contexts and discussed the unique challenges programming contexts present to studies of gesture. I offered an initial gesture taxonomy for computing education and suggested a research agenda to incorporate analyses of gestures in computing education.

Research on the representations available in activities and learning found that representations influence and structure how one attends to information and the cognitive processes used in the activity [26, 27, 28]. The representations help us remember, understand, reason, and communicate about the properties of and relations between objects represented in space. Moreover, findings in many scientific disciplines suggest that the usage of these representations is central to the learning of the discipline and its practice, helping one represent their knowledge and reason about abstract relationships [27]. Representations affect not only how much we know but how we know it.

It could be that CS might not be inherently spatial but spatialized because of how abstractions are represented. Liben argued that when a concept is abstract, it has a nonspatial referent, and as soon as that concept is given a form or location in space, it becomes spatial [29, 30]. She indicated that this also suggests that sometimes, the aspects of inherently nonspatial concepts can be spatialized.

The second perspective came from my interest in culture and systems of power. Research that focuses solely on correlations misses the complexities and nuances of how different people use spatial thinking [31, 32, 33, 34, 35]. Systematically minoritized and marginalized people tend to score lower on spatial ability assessments than white and Asian males do [36, 37, 38]. These same groups are also systematically underrepresented in CS. This correlation has started to be used to explain why such groups do not succeed in CS classrooms. We need to be careful and thoughtful about how we think about and analyze space; otherwise, we will create more gatekeepers to keep systematically underrepresented people out of CS.

Ramey reminded us that correlational studies using psychometric assessments typically only represent one small sliver of the diverse array of spatial thinking [31, 32]. She suggested that by using only these exogenous methods of assessing spatial thinking and learning, we are substantially missing the richness of what students are actually doing. Ramey described how endogenous accounts of spatial thinking are more asset based and are beneficial for capturing the spatial thinking and learning of marginalized and minoritized students, who typically underperform on psychometric assessments; however, they may still use spatial thinking [34, 33].

Taken together, these two perspectives led me to focus on theoretical pluralism [18, 19, 39]. No single theoretical framework will answer all the questions. Theoretical pluralism allows for a complete picture of how CS learning appears. Aside from examining correlations, we need many avenues of investigation.

1.2 Research Goals and Questions

As previously mentioned, the goal of this dissertation is to document how space influences and appears in CS learning. Specifically, my work aims to create a data-grounded theory of how teachers use embodiment to communicate and explicate computation and how students use embodiment to solve problems and make sense of computation.

I draw on constructivist, situative, and distributed learning theories to frame my investigation of space in CS learning [40, 41, 42, 43, 44, 45]. This framing involves certain assumptions about the nature of thinking and learning. From constructivism, I draw the notion that students are active agents in the construction of their knowledge. Students build their own unique representations and incorporate and connect new information with their preexisting knowledge. From situative theories, I consider learning to be situated within particular sociocultural contexts and activities. I also consider learning to be a social, communicative process and a process of consent into disciplinary practices. From distributed learning perspectives, I draw on the notion that thinking and learning are distributed processes involving the mind, body, other people, representations, objects, and other artifacts and tools available to someone within a specific context.

I use a grounded theory approach for data collection and data analysis. Grounded theory is a methodological approach for collecting data, analyzing data, and developing theory [46, 47, 48]. In a grounded approach, one iteratively collects data to develop theories and codes that data with the aim of developing a theory that fits the data. I have three guiding methodological principles inspired by this grounded approach. The first one is to ask basic questions. Before making broad claims about how students make sense, the factors that mediate learning, and the skills and types of reasoning that matter, basic questions must be answered. The second principle is to analyze multiple settings. The third principle is to analyze multiple perspectives, including perspectives of teachers and students and what resources they use.

1.2.1 Research Agenda

Conceptual Framework of the Kinds of Embodiment Teachers Use

I conducted a grounded analysis of a set of naturalistic video recordings of university professors teaching recursion in their classrooms. I wanted to understand how teachers use space in CS classrooms. Therefore, I looked for spatial representations, including

iconic gestures, spatial language and spatial metaphors, and artifacts, such as sketches or diagrams. While conducting this analysis, I noticed an interesting relationship between space and embodiment. The gestures the teachers created about computation seemed to be enacted metaphors; that is, the gestures seemed to act out a metaphor. Similarly, the teachers used spatial language (e.g., then, it goes down here; then, I jump up to the if-statement) while describing a code trace and used interesting language, such as calls, runs, and bound, to refer to function invocation, program execution, and variable assignment, respectively. However, all these are metaphors. The computer is not going down or calling anything, but teachers used metaphors and physicality to represent the abstract. Moreover, teachers metaphorically construe themselves as executing lines of code or constructing an agent who does that. Embodiment seems to play a central role in the ways teachers communicate information and meaning. This study sets forth the conceptual framework of the kinds of embodiment used to analyze the rest of the work.

A Critical Analysis of How Well Students Might be Able to Interpret Teachers' Embodiment

In this study, I considered learning by reflecting on how teachers communicate and structure learning opportunities—as a way to understand what students need to interpret in order to learn. Using the same data corpus from the first study, I sought to understand how well students could use or interpret the embodiment the teachers created and communicated. I focused this examination on gestures and metaphors, as only one teacher created a sketch. In my entire data corpus, embodiment was the only representation that the teachers used to make computation, physical. I found that while embodiment could have pedagogical significance, the ways teachers use it could be a source of confusion and add to the list of difficulties that make learning to program hard. Teachers randomly switch perspectives, refer to lines of code as if they exist in the real world, and create gestures that are either communicating important information or could also be nonsense. Professors use metaphors, such as calls and runs, without explicitly stating what they mean. The em-

bodiment, specifically metaphors, do not seem to be intentionally designed for explanatory power.

Students Use of Conceptual Blends and Embodiment to Support Their Sense-Making

Finally, I video recorded 10 dyads of college students while they solve recursive problems. While conducting another grounded analysis, I found that the students creatively constructed different metaphors and overlaid metaphors to make sense of computation. I used conceptual blending as a theoretical framework to describe the ways in which they blended multiple metaphors that allowed them to make sense of computation, to reason about the behavior of code and make predictions or explanations. I conclude by making a case that it is not about designing an everlasting representation that always helps one understand this concept. It is about finding a cognitive toehold on understanding this problem right now.

The analyses conducted are largely linguistic. I present my findings in the style of detailed episodes; this work is meant to be descriptive and interpretive. I provide my findings as episodes not only to present evidence to readers but also to enable them to come to their own conclusions, which might be contradictory to the ones I make. Moreover, as this work is interpretative, I do not provide exhaustive lists or categorizations.

1.3 Dissertation Overview

Chapter 2 provides a brief overview of recursion and literature review about what CSEd currently knows about how students learn recursion.

Chapter 3 unpacks the conceptual framework guiding data collection, analysis, and writing, and rationale for using these guiding theories.

Chapter 4 surveys the relevant literature in CER of the complexities associated with learning to program and sheds light on how little is known about embodiment in CSEd.

Chapter 5 presents a conceptual framework of the different forms of embodied repre-

sentations found in CS teaching.

Chapter 6 considers how professors orchestrate opportunities for learning, how well a student might be at interpreting teacher's embodiment and considers what might make interpretation more fluid.

Chapter 7 describes how students use embodiment and metaphor to make sense of computation. Specifically, I use conceptual blending as a theoretical framework to describe how students overlay multiple metaphors to make predictions about code behavior and explanations.

Chapter 8 summarizes the findings and implications, and avenues for future work.

This dissertation is written in the style of a multiple manuscripts dissertation. Each analysis chapter describes data analysis and collection, and reviews any relevant literature unique to the chapter.

1.4 Positionality

Because of the qualitative research approach and the fact that the work is meant to be descriptive and inductive, some background, positionality, and reflexivity about myself will help contextualize the findings in this dissertation. I was raised in an upper middle-class Black household. I have bachelor's and master's degrees in computer science. Because I am a computer scientist, this work was done from an insider perspective. It did make it difficult for me to identify some metaphors or to even describe some metaphors because that language for me is just what it is—it is not a metaphor. Someone without an outsider perspective may do a better job with that. However, being an insider made it easier for me to understand what people were talking about and to see meaning in some gestures. Because of this, I could easily be adding extra meaning to things than I meant to.

I also believe that if we are not actively accounting for systems of oppression, whatever we do will end up constructing and reimagining them. This means that equity issues do not work in a silo. If one is not actively trying to make something anti-racist, it will default

to being racist. The same holds true for other systems of oppression. Therefore, I try my best to use equitable methodological approaches and equitable ways of understanding what learning is. For example, I consider when I was an undergraduate, I had trouble understanding many of the metaphors my professors used in my CS classrooms and some metaphors made me uncomfortable. While that issue is pedagogical, it is also an issue of equity, specifically inequitable communicative practices used by teachers.

CHAPTER 2
SETTING THE CONTEXT OF RECURSION

In one of Dijkstra's infamous keynotes, he mentioned that he was surprised that so many of his students had difficulty coping with the concept of recursion because he knew "that the concept of recursion was not difficult" [49]. However, recursion is a notoriously tricky programming concept for students to learn and teachers to teach. Decades of CER have consistently found recursion to be one of the concepts that students struggle with the most and where they have incomplete or incorrect understandings of the concept. Students typically have difficulty constructing viable mental models of recursive functions [50, 51, 22], and computing teachers have the daunting responsibility of scaffolding students' development of viable mental models of recursion [52, 53]. Without a viable mental model, students cannot accurately reason about recursive functions. Accordingly, how students learn to "think recursively" [54] and the best practices for teaching recursion have been subjects of CER for decades.

In this chapter, I describe findings about the difficulties that students have with learning recursion. I first explain what recursion is, describing some concepts relevant for understanding it. I then describe results about the viable and non-viable mental models that students typically form about recursion. Next, I tell a specific conceptual model (a stack-based notional machine) that teachers typically use to describe recursive invocation. I conclude by presenting an example of a recursive invocation.

None of this research has ever considered embodiment. In later chapters, I describe the ways in which teachers' use of embodiment could make learning recursion difficult and how embodiment makes recursion physical.

In this dissertation, the discussion of recursion applies to programming languages under the imperative paradigm, which includes C++, Java, and Scratch. While imperative

languages might be considered the most authentic in professional practice, they could limit the applicability of my findings.

2.1 Recursion in Programming—Recursive Invocation

A common definition of recursion is a function or subroutine that invokes itself. Invoking a function or function invocation is a command to the computer to execute a set of actions.

While this definition might seem like an endless cyclical loop, a well-defined recursive solution should never be infinite. Recursion is a technique used to break down a difficult problem into smaller subproblems that are easier to manage. Each subroutine should never solve the same problem. At some point, the smaller problem should be so simple that it can be solved easily, which ends a recursive invocation.

To clarify this, I describe a typical problem used to introduce recursion to students, which is calculating the factorial of a number (e.g., 5! [5 x 4 x 3 x 2 x 1 = 120]). 5! is the same as 5 * 4!. 4! is the same as 4 * 3! and so on. The smaller problem that is easier to solve is (n-1)!. Therefore, the factorial of some value n (or n!) can be obtained by multiplying n times (n-1)!. The smallest instance of that problem is 0! or 1!.

Figure 2.1 is an example of the function factorial that computes the factorial of an inputted value in code. Although the complexity and structure of a recursive function can vary, a recursive solution is typically expressed using an if-else statement, which tells the computer to execute lines of code if a specified condition is true. In this example, the if-statement (if number = 0) defines the base case or the stopping criterion that terminates the recursive invocation. The base case should produce a trivial result without recurring. Again, the base case is the smallest instance of the problem, which is 0! or 1!.

The else statement is the recursive case, which is the recursive invocation in which the function breaks down the problem to solve part of the original problem [55]. A recursive function can have multiple base cases and recursive cases.

Recursion can range in complexity. The factorial function is an example of a tail-

```
public int factorial (int n) {
        if (n == 0) {
                return 1;
        }
        else {
                return n * factorial(n - 1);
        }
}
```

Figure 2.1: The code for the function factorial.

recursive invocation, which means that there are no lines of code following the recursive invocation. Another type is the non-tail end, which has lines of code after the recursive call, and these lines of code are executed after the recursive execution terminates.Figure 2.2 is a representation of typical non-tail end execution.

2.2 Recursive Execution

In this section, I describe the mechanisms behind a recursive execution, that is what happens when a computer executes the set of actions. To understand execution, I first describe a stack-based notional machine.

Researchers have described several strategies teachers use to help scaffold students' development of recursive mental models. Wu et al. explain that it is a teacher's responsibility to present instructor defined conceptual models to help facilitate the learning of recursion and help students develop their own viable mental models [52]. Wu et al. described the most widely used models, reported in Table 2.1 [52].

Teachers might also use a specific kind of conceptual model in computer science education (CSEd), known as a notional machine. It is widely accepted in CER, that teachers need to scaffold students' mental models of a notional machine.

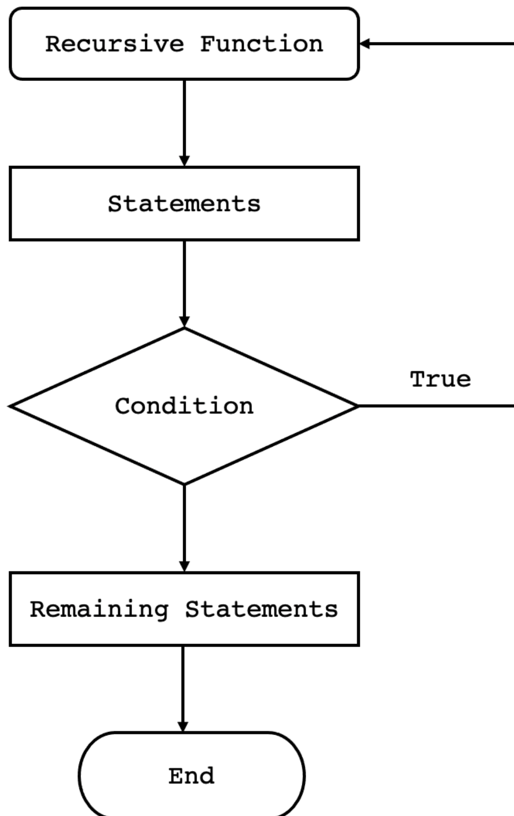


Figure 2.2: Non-tail end execution

Recursive execution is invisible, and the code does not make it evident how recursive execution happens. For example, in the previously mentioned figure with a non-tail end recursive function, it's not evident that after the recursive execution finishes, all the lines of code after the recursive statement get executed. Therefore, having a working notional machine or mental model of recursion helps students have a better understanding, reason about, and conceptualize how a recursive function executes. Notional Machines are representations that highlight, "those things that are important to look at, or that do something that makes apparent otherwise invisible behavior which, if un-noticed or misunderstood, would cause the learner to go hopelessly wrong" [56].

Notional machines can vary in complexity but, one a simple version is to use an analogy to describe some programming construct or concept. For example, students are typically taught that recursive execution works like a stack (See Figure 2.3). In CS, a stack is a

Table 2.1: Most commonly used conceptual models. Table adapted from [50].

Concrete Conceptual Model	Abstract Conceptual Model
Counting using nested Russian dolls	Recurrence relations using factorial as an example
Design recursive algorithms using a block diagram tracing technique	Identify base and recursive cases
Verify recursive algorithms using a block diagram tracing technique	Design and verify recursive algorithms using induction

metaphor that represents a kind of data structure of a collection of elements. Stacks work as LIFO (last in, first out) and have two main functionalities: (1) pushing, or adding elements into the stack, and (2) popping, or removing elements in the stack. A similar analogy for a stack is to think about stacking plates, each plate goes on top of each other, with the first plate at the bottom and the last plate placed at the very top. When we take plates off the stack, we start by taking the topmost plate - or last plate added - in the stack.

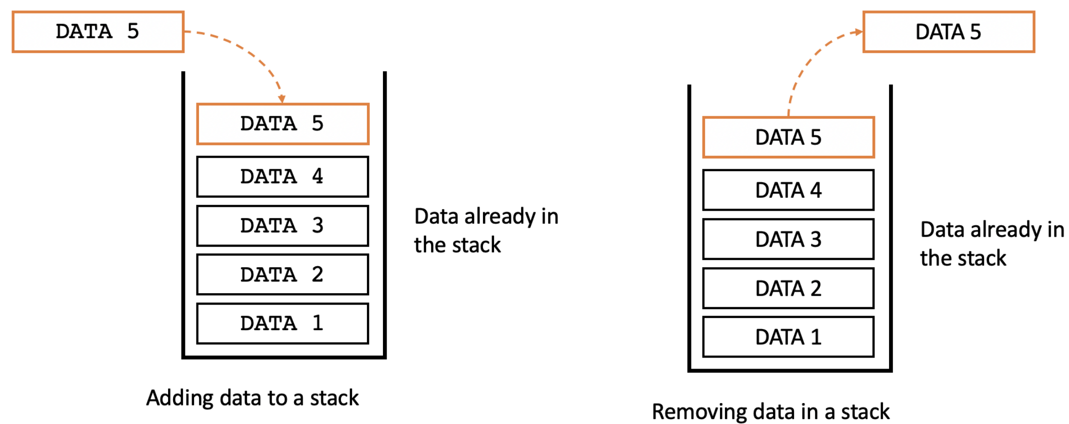


Figure 2.3: A representation of a stack in CS. The first image shows elements pushing into the stack. The second image shows elements popping out of the stack.

A recursive solution cyclically iterates and evaluates each smaller problem and then combines the results. In a recursive execution, every subroutine "has its own private, fixed working space" [57]. Similarly, then, each subroutine is "pushed" onto the stack halting execution until the base case is executed, terminating the recursive execution. At which point, each subroutine is "popped" off the stack and continues execution.

There are different notional machines to represent recursive execution. For example, Wilson et al. describe a “lesser-used” notional machine, algebraic substitution, ”when a function is applied to arguments (the ”actual parameters”), first the arguments are evaluated, then all instances of the function’s formal parameters are replaced with their argument values. The resulting program is then evaluated using these concrete values” [22]. Since stack was the only notional machine used in my research, I will focus primarily on the stack-based notional machine.

To illustrate the execution process, Figure 2.4 is a representation of how the computer interprets and executes the invocation factorial(5) using a stack-based notional machine. For this code trace, a computer invoked the factorial function with a value of 5 - factorial (5). Simply put, see Figure 2.4, the computer first checks to see if the input value is equal to n. If it is, the function returns 1, stopping the recursive invocation. However, if the value is not equal to n, then the same function is invoked again, but with one less than the original, input value.

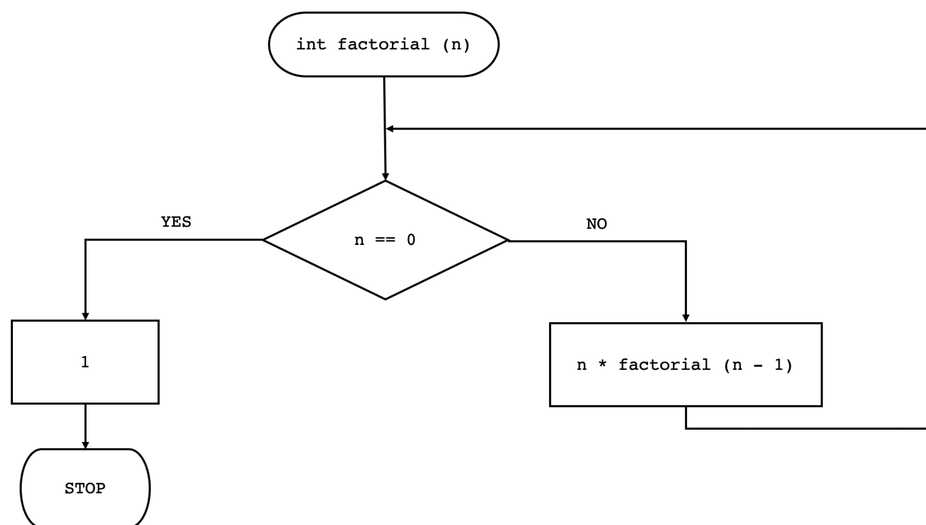


Figure 2.4: A representation of a flow chart of the factorial function

If the computer executes factorial(0), the function returns 1 and never hits the recursive case.

Each invocation forms its own execution frame that is placed onto the stack, see Fig-

ure 2.5. With recursion, the computer waits for return values coming from other execution frame. These other frames are higher up the stack. When the last item on the stack finishes execution, that context generates a return value. This return value gets passed down as a return value from the recursive case to the next item.

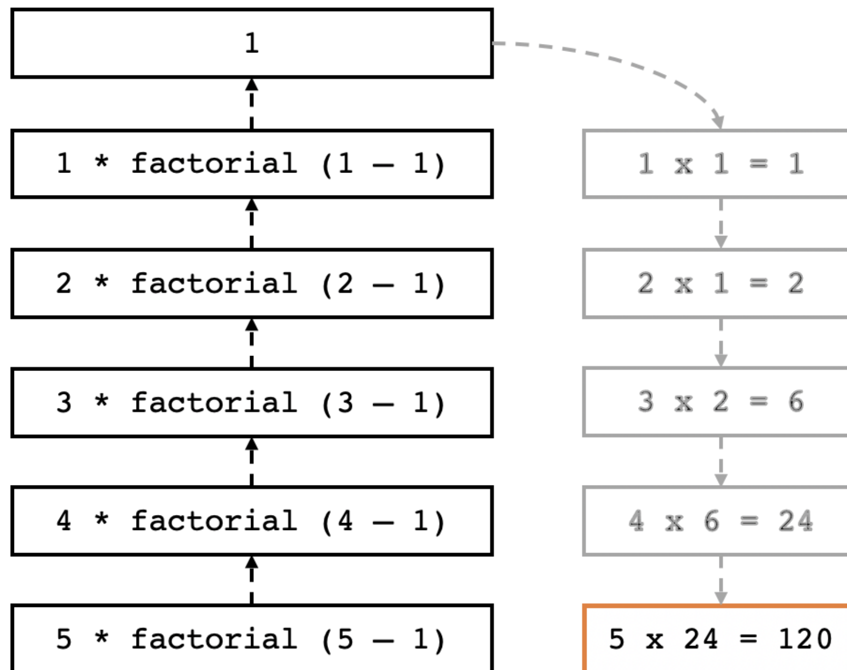


Figure 2.5: A representation of a code trace of the factorial function

1. The computer invokes factorial() with 5 as the argument passed. The base case is false, so the recursive case is executed. The invocation factorial (5 - 1) is placed into the stack and suspended.

2. The computer invokes factorial() a second time with 4 ((n - 1)) as the argument passed. The base case is false, so the recursive case is executed. The invocation factorial (4 - 1) is placed into the stack and suspended.

3. The computer invokes factorial() a third time with 3 ((n - 1)) as the argument passed.

The base case is false, so the recursive case is executed. The invocation factorial (3 - 1) is placed into the stack and suspended.

4. The computer invokes factorial() a fourth time with 2 ((n - 1)) as the argument passed. The base case is false, so the recursive case is executed. The invocation factorial (2 - 1) is placed into the stack and suspended.

5. The computer invokes factorial() a fifth time with 1 ((n - 1)) as the argument passed. The base case is false, so the recursive case is executed. The invocation factorial (1 - 1) is placed into the stack and suspended.

6. The computer invokes factorial() a sixth time with 0 ((n - 1)) as the argument passed. Now, the base case is true, so return 1.

At this point, the computer has decreased the argument by one on each function call until the condition to return 1 is met.

1. From here the last invocation completes, so that function returns 1.

2. Next value is 2, so the return value is 2. (1×2).

3. Next value is 3, so the return value is 6, (2×3).

4. Next value is 4, (4×6). 24 is the return value to the next context.

5. Finally, value is 5, (5×24) and we have 120 as the final value.

2.3 Learning Recursion

Learning to write and trace recursive functions is challenging for multiple reasons [55], including that people do not naturally think recursively and that there are no obviously recursive real-world experiences or analogies that students can draw upon to understand recursion. Another reason is that most students have not “previously encountered neither

the concept nor its associated vocabulary" [50]. Computing education, programming languages, and software development all rely on metaphors. The terms "recursion" and "function" are conceptual metaphors used to name abstractions [58]. Many of these difficulties with teaching and learning resulting from the run-time phenomena of recursive procedures are not visible and are also difficult to describe. Pirolli and Anderson argued that this results from the lack of everyday analogies that can bridge the digital world with the analog world and from most students not having previously encountered either the concept or its associated vocabulary [54].

Arguably, one of the main challenges is that students have trouble understanding recursive execution because the code and computer hide the execution of a recursive function, which is a difficult process to describe [55]. To write and trace a recursive function, students must understand the following concepts: function invocations, function returns, parameter name reuse, and a function calling itself. Students are expected to just trust the hand-waviness or magic of recursive invocation. In a survey of students' and tutors' perceived difficulty with programming concepts, Milne and Rowe concluded that "these concepts are only hard because of the student's inability to comprehend what is happening to their program in memory, as they are incapable of creating a clear mental model of its execution" [59]. Likewise, George argued that students have trouble understanding recursion largely because they fundamentally do not understand subroutine execution, "don't understand what really happens when a function is called, they don't have the proper foundation required to build an understanding of recursion" [60].

2.4 Mental Models of Recursion

Studies have found that even after substantial training, students still lack an understanding of the mechanism of recursive execution. When describing why students get recursive problems wrong, research typically attributes it to students having an undesired mental model. Researchers who identified error patterns in students' recursion evaluation traces

attributed the errors to students' maladaptive mental models of recursion [55]. Mental model theory tells us that people construct and refine mental models to reason about and predict events happening in the world. However, mental models do not necessarily reflect events accurately or completely because of a person's own biases and misunderstandings.

Researchers have explored the different mental models that both experts and novices have developed about recursion [53, 61, 55]. Kahney's list of mental models of recursion is arguably the most well-known and cited in computing education scholarship [53]. Kahney's work has since been extended by Gotschi et al., who identified the mental models that were likely to lead to a correct understanding of recursion [61]. Table 2.2 lists the different mental models and the descriptions of each.

Table 2.2: Mental models of recursion. Adapted from [50]

Model	Usefulness	Definition
Copies	Viable, expert view	Recursive procedures are seen to generate new instantiations of themselves, passing control, and possibly data forward to successive instantiations and back from terminated ones (same as Kahney)
Looping	Flawed	Recursive procedure is viewed as a single object having an entry point and an iterative action part. Function variables are treated as global. This erroneous model does not consider multiple instantiations of a procedure and does not account for the undoing of a sequence of recursive calls (same as Kahney)
Active flow	Flawed	Students understand recursive computation until a base case is reached, but do not understand the computation "unrolls" itself
Step	Not viable	Students have no model of recursion and only comprehend a small portion of what is going on such as evaluating the if and else clauses only once
Return value	Flawed	It is assumed a return value is evaluated at each call and the collection of return values are combined to provide a solution
Syntactic or magic	Not viable	Students view recursion as consisting of a specific surface structure, a template including base and recursive cases, with little realization of recursion as a problem-solving technique (same as Kahney)
Algebraic model	Flawed	Implementation of a recurrence relation is viewed as an implementation of an algebraic problem
Odd	Not viable	Model has aspects of copies or looping models but not accurately enough to be able to predict the behavior of recursive code (same as Kahney)

Kahney found that experts have a model of recursion that "defines recursion as a pro-

cess that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones," referred to as the copies model [53]. This mental model is in line with the description of the factorial invocation and execution described in the previous section, in which each invocation creates its own execution frame that is halted or suspended until the base case is invoked.

However, students typically have flawed models of recursion, such as the looping model, in which recursion is considered synonymous with iteration or looping. In this model, a recursive invocation does not create its own frame; instead, the same frame is repeated but with different values [55]. The process is repeated until the base case is invoked. Chao et al. described this model as flawed because it typically results in a lack of understanding of the passive flow, that is, the process that happens after the base case is invoked [55].

Figure 2.6 was adapted from Chao et al. and illustrates the copies and looping mental models using the aforementioned factorial function [55].

Factorial Method

```
public static int f(int n){
    if (n == 0) {
        return 1;
    }
    else {
        return n * f(n - 1);
    }
}
```

Multiplication Equations Method

```
public static void g(int m, int n){
  int product = m * n;

  if( n != 0 ){
     g( m, n - 1 );
     System.out.println( m + " * " +
              n + " = " + product );
  }
}
```

Copies model

(a)

```
f (3)
| 3 * f (2)
| 3 * | 2 * f (1)
| 3 * | 2 * | 1 * f (0)
| 3 * | 2 * | 1 * 1
| 3 * | 2 * 1
| 3 * 2
6
```

(b)

```
g (10, 3)
|   g (10, 2)
|   |   g (10, 1)
|   |   |   g (10, 0)
|   |   "10 * 1 = 10"
|   "10 * 2 = 20"
"10 * 3 = 30"
```

Looping model

(c)

```
f (3)
3 * f (2)
    2 * f (1)
        1 * f (0)
            1
```

(d)

```
g (10, 3)
30  g (10, 2)
    20  g (10, 1)
        10  g (10, 0)
            return nothing
```

Active model

(e)

```
f (3)
3 * f (2)
3 * 2 * f (1)
3 * 2 * 1 * f (0)
3 * 2 * 1 * 1
6
```

(f)

```
g (10, 3)
30  g (10, 2)
30  20  g (10, 1)
30  20  10  g (10, 0)
30  20  10  return nothing
```

Figure 2.6: Copies and looping mental models of the factorial function [55]

However, Chao and colleagues argued that attributing error patterns to mental models implies that mental models are “relatively stable, integrated knowledge structures,” when we know that students can fluently switch between mental models while working on the same problem [55]. Using diSessa’s knowledge in pieces as a theoretical framework, Chao argued that students rely on loosely coordinated knowledge elements to dynamically construct mental models of recursion. Moreover, Sanders et al. found that about half of the students who used the copies mental model on one task switched to a different mental model on another task [62]. Unlike those of experts, students’ mental models are not static. Bettin argued that “learning is a messy process: we work to adapt models, gain new information, connect that information, and often forget, misremember, or misapply other information” [63]. A sign of expertise is the formation of stable mental models: “By contrast, experts’ mental models are more stable and accurate, and draw on general principles rather than superficial characteristics. A challenge of programming education is to facilitate the evolution of students’ models so that they have these features” [64].

CHAPTER 3
CONCEPTUAL FRAMEWORK

This chapter provides details about the conceptual framework that I use to analyze how embodiment influences and appears in CS learning. As previously mentioned, this conceptual framework was developed from a grounded analysis of space in CS learning. In some ways, finding a relationship with space and embodiment should have been expected. In STEM education literature, the relationship between embodiment and spatial thinking has been documented. For example, DeLiema and Steen describes how students use gesture to offload their spatial thinking [65]. In a literature review about how embodiment could be used to teach spatial thinking, DeSutter and Stieff describe how, "the body and embodied knowledge to represent and think spatially has also been identified among expert STEM professionals engaged in their discipline [66]. Ethnographic studies of scientists engaged in authentic practice have found that complex spatial ideas are often conveyed using representational gesture-based and body-based metaphors" [66].

In the rest of this chapter, I first describe embodiment as it is used in this dissertation. I then describe three different embodied representations that I used to investigate embodiment. These representations were the most salient in my research.

3.1 Embodiment

Embodied perspectives on cognition and learning (e.g., [67, 68, 69] have developed with the insight that we cannot study cognition as a phenomenon that is isolated in individual human minds. Instead, cognition has to be regarded as interaction between body and mind and between individuals and their material and social environments. Indeed, Almjally and colleagues argue that such perspectives have provided useful insight for STEM learning, where disciplines depend on, "representational systems that require sensory encoding, such

as data visualization, and rely on high abstraction, such as mathematical formulae or programming code" [70]. Research in mathematics and science learning has highlighted the affordances that body-based strategies may provide for teaching and learning [6, 28, 71, 72, 73]. Moreover, there is evidence that adopting embodied approaches, such as using object manipulation or gesturing in education, can not only enhance a student's ability to connect the abstract to the concrete, but also improve memory and cognitive skills, such as strategic or spatial cognition and the reasoning abilities used in problem-solving [74, 75].

However, there is a lack of consensus on the ways "embodiment," "embodied," and even "embodied cognition" are used. For example, in a 2012 special edition of the Journal of Learning Sciences, Stevens describes two more orienting perspectives of embodiment [6]. The first is the conceptualist perspective, which comes from cognitive linguistics. This approach assumes that we all have shared experiences and thereby have common mental schema: "ideas are organized in conceptual systems grounded in physical, lived reality" [6]. The second is the interactionist perspective, which came about from studying cognition and learning within specific sociocultural and sociomaterial contexts. This perspective understands, "the body as an interactively organized and public resource for thinking, learning and joint activity," that produces meanings and actions through diverse means, including tool use, gesture, pointing, prosody, intonation, physical orientation, gaze and talk" [6].

In general, embodiment implies that the mind is not the sole source of knowledge, but we make meaning about the world from our body-based, lived experiences [6]. This means that the body and body-based experiences are active in meaning-making, sense-making, and, importantly, including tool use, gesture, pointing, prosody, intonation, physical orientation, gaze, and talk. Furthermore, learning is seen as "depend[ing] crucially on our bodies, especially our sensorimotor apparatus, which enables us to perceive, move, and manipulate" [75].

In this dissertation, I define embodiment to encompass three different aspects of body-based meaning [75]:

1. the grounding of the abstract domains within concrete domains and “the embedding of the body in a deeply situated sense-act-process loop to make and represent meaning” [75]

2. the physical body and its sensorimotor capabilities

3. the use of the tools to extend the body’s capabilities

Such aspects manifest in many embodied representations, including metaphors, gestures, sketching, winking, changes in voice intonation, and facial expressions. However, I investigate embodiment by examining how teachers and students use embodied representations, specifically metaphor, gesture, and sketching. Besides being the most salient, I chose to focus on those three because of their practical applications for pedagogy [75]. That is, one can easily design a metaphor for explanatory power, for example, but cannot replicate more fine-grained embodiment.

3.2 Metaphor

Since the mid-twentieth century, philosophers have shown that metaphors permeate all discourse, are fundamental to human thought, and are pedagogical tools. Similarly, one of the most important revelations is that metaphor is not merely a linguistic phenomenon but also a fundamental principle of thought and action: ”Metaphors aren’t just ways of talking, they are ways of thinking” [76]. Similarly, researchers have been attracted to the potential contribution of metaphors for learning.

Lakoff and Johnson proposed Metaphor Theory, where they argued that we use metaphors to talk, reason, and think about abstract domains like space, time, love, and mathematics. Metaphors concretize abstract entities, by mapping the abstract “space” to something concrete [77]. Metaphors make “apprehending” abstract concepts “accessible” to students through comparison with familiar, bodily experienced concrete concepts.

Embodiment is a central assumption of Metaphor Theory. Lakoff and Núñez state that language, specifically conceptual metaphor, is a product of embodied thought: "much of what is 'abstract'... concerns coordination of meanings and sense-making based on... forms of metaphorical thought. Abstract reasoning and cognition are thus genuine, embodied processes" [77]. For a mental representation to be "embodied" in the sense most commonly invoked by metaphor researchers, it must be instantiated at least in part by a simulation of prior or potential bodily experiences, within modality-specific components of the brain's input and output system.

To grasp abstract concepts, we project embodied or sensory motor reasoning onto them. Affection, for example, is frequently understood in terms of physical warmth: ""She was cool to him"; "he gradually warmed to her"; "they had the hots for each other."

Lakoff and Nunez claim that the mathematics we used to describe as disembodied is in fact embodied [67]. Humans use their bodies, mind, and brain to both form and understand mathematics. All mathematical content resides in embodied ideas and many of the most basic, as well as the most sophisticated, mathematical ideas are metaphorical. Their argument is that all mathematical concepts arise as metaphors where mathematical ideas are ways of 'mathematicising' ordinary ideas and these metaphors are in turn grounded in, or based on, our embodiment.

They argue that our understanding of algebraic variables is similar to our understanding of pronouns: "Whoever did this was sick" should be compared to "If X + 2 = 7, then X = 5". Empirically they demonstrate some of their insights by discussing the gestures mathematicians typically use to explain their ideas. Therefore, not only categorization is grounded in (shaped by) the body, but so is cognition in general, including spatial and social cognition, problem solving and reasoning, and natural language.

Moreover, metaphors may indicate how we conceptualize different mathematical concepts. For example, "take away" and "break" or "count up" and "next" may indicate how we conceptualise numbers as collections of objects or as points along a path [67].

Of particular interest here is that many metaphors involve mappings from concrete domains to abstract domains such as time, cause, depression, and love for which no (or little) experience-based schematic structure can be described. This phenomenon extends to the language of science and mathematics. Amin found different metaphors implicit in the scientific use of the term energy, including construing energy as a substance transferred from one entity to another, energy as a whole object composed of parts, forms of energy were construed as containers, and others [78].

Metaphors involve the mapping between two conceptual domains.

1. The first domain is the source domain, or the conceptual domain from which we draw metaphorical expressions.
2. The second domain is the target domain, or the conceptual domain that we try to understand.

Wilbers and Duit argue that from a constructivist point of view, this mapping or similarity is constructed and dependent on prior experience with source and target [79]. They describe how this mapping could create difficulties for student learning. Moreover, understanding metaphors can be difficult. Metaphors should arguably be framed as cultural tools; they are products of specific sociocultural environments and are historically constructed.

> if a teacher generates a metaphor or analogy, the target domain is as well known to them as is the source domain. The student is instead totally ignorant of the scientific concepts that are communicated. Therefore, the teacher's use of the metaphor or analogy is presumably different from the student's: while the teachers construct the similarities, the students have to search for them.

3.2.1 Metaphors support Understanding and Reasoning

The question that emerges from such analyzing metaphors is the extent to which the metaphors are resources for understanding and reasoning. Metaphors have been acknowledged for

their capacity to help with learning abstract concepts. Manches [80] argues that despite metaphor theory critiques, it has offered an explanation for our ability to think and reason about abstract concepts and has provided a valuable theoretical framework for examining the conceptual underpinnings of abstract concepts, notably in science (e.g., thermodynamics [81, 82]).

Similar to analogies (see [83]), metaphors involve mappings between a source domain (e.g., movement) and target domain (e.g., time). Both analogies and metaphors express comparisons and highlight similarities, but they do this in different ways. The key difference, according to Brookes and Etkina, is that analogies suggest the source domain is like the target domain (e.g., life is like a box of chocolates) [81]. Whereas a metaphor suggests the source domain is the target domain (e.g., they are a shining star). In a study about physicist uses of metaphors, Brookes and Etkina say that physicists need to assert something more than ”is like” they need ”is” in their reasoning process [81]. They conclude that “is” is a fundamental trait of how knowledge is generated and represented in physics.

Moreover, Brookes and Etkina add that metaphors encode analogies, which supports reasoning about a problem or a solution. Physicists are able to use these metaphorical systems to reason productively about a particular situation or problem. For example, the electron is a wave metaphor can be used productively to explain the Heisenberg uncertainty principle: “I often think of it. . . in terms of Fourier transforms and the reciprocity between the bandwidth of the channel and the length of the signal pulse that can be detected” [81].

Duit argues that metaphors compare without doing so explicitly [84]. They describe this point using the metaphors of calling education “sheep herding” or a teacher “the captain of the ship.” Taken literally, these utterances are “absurd,” but metaphorically, the utterances grounds our understanding and “provoke anomaly” and “surprise.”

They continue that the “surprise” or “anomaly” aspect of metaphors is what makes them significant in the learning process: ”Something happens to us when we first read a fresh metaphor. We are reorganizing our patterns of previously organized meaning” [84].

Metaphors allow new perspectives and help us to see the familiar in totally new ways. Duit uses the argument from Gowin to explain this surprise and anomaly. Gowin describes the metaphor ”A paintbrush is a kind of pump,” which is, initially, a surprising statement because it provokes thought and causes one to construct an analogical relations that gives the statement meaning [85]. This ”generative power” of metaphors makes them potentially valuable tools in conceptual change learning. They provide what is essential to this aspect of learning, namely making it easier to restructure the already known and familiar.

In similar vein, in Roschelle's seminal research on convergent conceptual change, Roschelle describes students using a pulling metaphors to construct their explanation of acceleration (i.e., “acceleration pulls the tip of the velocity vector”) [17]. This use of pulling contrasts with the common misconception “force as a mover,” which specifies force as directly related to change of position. The key transformation of conceptual structure for successful students was the redirection of pulling from “acceleration pulls the particle” to “acceleration pulls the tip of velocity” a transformation of a “misconception” into an explanation that is more compatible with the interpretation of acceleration in scientific practice.

3.2.2 Metaphorical Construals

Dreyfus et al. provided evidence that experts and learners talk and reason about emergent processes as if they were material substances [86]. Ackermann refers to this as perspective-taking to be a body-based activity, necessary for acquiring a deeper understanding [87]. Ackerman uses the language of “diving in,” or situating oneself to become part of the phenomena, and “stepping out” of the phenomena, which helps someone reflect on their experiences, forming more abstract insights.

In their seminal work, Ochs et al. found that scientists were, “taking on the perspective of (empathizing with) some object being analyzed and by involving themselves in graphic (re)enactments of the physical events” [88]. Ochs et al. described these linguistic constructs “where the participants moved between a normative scientific description of a phenomenon

to more personal 1st person description as liminal worlds, because they were episodes in which objective facts were blended together with subjective reasoning from a first-person perspective [88]. These liminal worlds created a qualitatively different set of resources from which to reason and were found to be productive in model and theory building."

From a pedagogical perspective, some work has referred to these metaphorical construals as 'role-playing' [75]. Some roleplay involves students pretending to be other actors, like scientists or politicians in order to better understand the perspective of another haman. However, another kind of roleplay, the kind referred to in this dissertation, has students play as entities to simulate, "the processes of natural phenomena and clarify students' interpretation of these phenomena" [89]. Roleplay is used as an analogy, which involves students actively reinterpreting information. This enables students to generate a deeper understanding and draw on their own experiences to make sense of new information. Aubusson and colleagues state that the key feature of roleplay lies in its ability to help students develop and create viable mental models, which can then be used to create theories to explain phenomena [89].

3.3 Gestures

Gesture has been studied to understand its role in teaching and learning in several domains (e.g., mathematics, physics) with a research lens informed by embodied cognition, learning sciences, and educational psychology. In this paper we focus on the study of gestures from a learning sciences and educational psychology perspective with an emphasis on disciplinary education research in mathematics. We limit our focus first, because mathematics offers the tightest connection to computing and second, because the literature on the use of gestures in learning is expansive, and thus our limited focus helps set boundaries for this initial exploration of gesture research to computing education.

3.3.1 Gestures Defined

While there is a common perception of gestures, within the learning literature characteristics of gestures have been described to establish a shared understanding. Gestures are defined as visible, external representations of what people are thinking [90]. They are spontaneous hand movements produced when talking [91]. Yet, while gestures are hand movements, not all hand movements are gestures [92]. Gestures can be distinguished from other hand movements by four characteristics [92, 93, 94]:

1. Gestures begin from a position of rest, move away from this position, and then return to rest.
2. Gestures have a peak structure, also referred to as the stroke, which is generally recognized as a moment of accented movement to denote the function of meaning of a movement.
3. The stroke phase is preceded by a preparation phase and succeeded by a recovery phase in which the hand and arm move back to their rest position. Consequently, gestures have a clear beginning and ending.
4. Gestures are often symmetrical.

3.3.2 Gesture Can Reveal What a Learner Knows

Gestures have been found to be beneficial for instruction and understanding student knowledge. Gestures produced while students explain their reasoning provides unique insight into their thought processes [95]. For example, Novak and Goldin-Meadow describe a child who believes that the amount of water changes when it is poured from a tall, thin container into a short, fat container which indicates the child does not understand the concept of conservation of liquid quantity [95]. The child justified their belief by saying, "this one is taller than this one," while making a C-shaped gesture to indicate the narrow width of the tall container, followed by a wider C-shaped gesture to indicate the larger width of

the short container. The child is highlighting one dimension of the containers in speech (height), but his hands make it clear that he is beginning to think about a second dimension (width). Their gesture is conveying different information than their words. When someone produces different information in gesture than in speech, it reveals that they know more than they say. The information a learner conveys uniquely in gesture is often implicit knowledge, not yet accessible to explicit understanding [28].

3.3.3 Gesture Production Can Support Learning

Gesture may provide an avenue through which learners can consider new ideas. Broaders et al. found that children told to gesture added novel strategies to their repertoires that were found only in gesture [96]. These children were also more likely to profit from instruction in math. After the lesson (when they were no longer gesturing), the children were able to solve math problems on a paper-and-pencil test that they could not solve before the lesson.

3.3.4 Seeing Gesture Can Support Learning

A number of studies have identified that students are more likely to profit from instruction when a teacher gestures [97, 98, 99]. Ping et al. found that children who received instruction with gesture improved more than children who did not [100]. This might be because of gesture's ability to ground the abstract language of the lesson to the concrete physical environment and gesture's ability to convey ideas through "its representational form" [100, 99]. Physical environment refers to the setting for the interaction, the interlocutors, the focal tasks, and the representations, tools, technological resources, and social dimensions [101].

3.3.5 The Connection Between Gesture and Learning

Previous research suggests gesture promotes learning by engaging motor systems and encouraging students to link abstract concepts. Gesture enhances spoken communication.

Ping found that the effects of gesture on learning stem from its capacity to engage the motor system [102]. Gesturing thus supports learning because it is a type of action. Yet, a gesture is a representational action and not an action on objects, which is typically intended to carry out specific functions. This difference is responsible for gesture's unique effects on learning. Action on objects leads to shallow learning since it tends to lead people to think that their learned actions are relevant only to those objects [103]. This may hinder generalization by focusing learners on details that get in the way of transfer. In contrast, gesture, which occurs apart objects, provides a "physical distance," which facilitates abstraction and generalization to new contexts yet still engages motor systems. This distinction is what improves learning [104]. Gesture leads to deeper and more flexible learning.

3.3.6 Gesture Taxonomy

Taxonomies of gesture classify gesture based on certain functions or models of gesture production [105, 106, 107]. McNeill's taxonomy is widely used in gesture studies on educational issues and is considered applicable to gestures "in any type of discourse or any content area" [92, 101]. For our analysis, we will use McNeill's taxonomy as a framework.

McNeill's taxonomy outlines four types of gesture: (1) deictic, or pointing gestures that indicate objects or locations, (2) iconic, which are gestures that depict semantic content directly via the shape or motion trajectory of the hand(s), (3) metaphoric, which depict semantic content via metaphor, and (4) beat, which are motorically simple, rhythmic gestures that do not express semantic content but that instead align with the prosody of speech [108]. Using mathematics as a domain, we will explore each of these types of gestures in more depth.

3.4 Tool Use: Sketching

Embodiment is also used to address the manipulation of objects. Objects are, "those things that are nameable, identifiable, stable, and can persist through time, such as pencils and

cars". Tools are a specific kind of object employed to alter or interact with other objects. However, as Ibrahim-Didi et al. argue, tool use as a form of embodiment is largely overlooked [75]. One type of tool use I consider in this dissertation is sketching, which was the only kind of tool use found in my data corpus.

Experimental psychologist James Gibson developed the ecological approach towards embodiment, which suggests that the body serves as a mediator between perception and action; both perception and action are influenced by the environment [69]. Affordances are important to this conceptualization, which are opportunities or possibilities for action placed in the environment; tools can serve as an affordance. For example, a graspable object with a rigid sharp edge affords cutting and scraping (a knife, a hand axe, or a chopper). Gibson had the following to say about tool use:

> The embodiment of objects and tools may be defined as the sense that those objects and tools has become "part of us" in a similar way that our limbs or our fingers are parts of us. Most of the literature concludes that people can extend the borders of the physical body to temporarily incorporate different prosthetics, such as rubber hands, into their body image (i.e., their conscious beliefs regarding their bodies; and certain external objects, such as tools, into their body schema (i.e., their unconscious knowledge of their bodies and its capacities. They are a sort of extension of the hand, reflecting the capacity to attach something to the body. The possibilities they afford are not restricted to motor or user-centered actions (graspable, portable; i.e., actions "on" them) but can also concern mechanical or tool-centered actions. "The short answer is that in addition to altering our sense of where our body ends each tool reshapes our "enactive landscape' – the world we see and partly create as active agents. With a tool in our hands we selectively see what is tool relevant; we see tool dependent affordances;"

Cognitive scientist David Kirsh explains that that tool use and embodiment are en-

tangled: ”...interacting with tools changes the way we think and perceive – tools, when manipulated, are soon absorbed into the body schema, and this absorption leads to fundamental changes in the way we perceive and conceive of our environments” [109]. Kirsh continues that tools become extensions or parts of our body and “body schema” [109].

This theory of tool use gives us another perspective to the importance of sketching in CS learning. While we typically only consider sketching to offload or distribute cognition, Kirsh [109] and Wright [110] suggest that the act of sketching might serve another purpose, that is thinking with a distorted model and embodying whatever is being sketched.

Fundamentally, the use of these tools change our perceptions and our capabilities of conceptions. Kirsh uses examples of dancers marking and mechanics sketching a model [109]. Specifically, to both of these examples, he argues that whether marking or sketching a model, these people are creating ”imperfect models” that provide cognitive support for their reasoning and doing. It helps them explore/consider principles ideas better than using an ”undistorted” model.

Kirsh argues that working with an “imperfect model” is advantageous because the creator gets to model only what they want to reason about and think with. Important to his argument is that with imperfect models, what is sketched or marked highlights the aspects the person is paying most attention to and the aspects they want to think about, which primes thinking. He asserts that, ”Accuracy is not important, flow is.”

Wright uses the metaphor of ”play” to explain the relationship between drawing and embodiment [110]. Through play, children portray and embody different thoughts, emotions, and actions.

Similarly, Sherin has described how it when analyzing sketches its important to analyze the process and not just the final sketch [111]. Research on how students construct sketches to problem solve and generate solutions, has shown that students’ sketches both reflect their understanding of the domain and scaffolds the problem-solving process. Sherin suggests that these self-constructed sketches reflect someone’s underlying conceptions [111]. The

process of generating and revising sketches also helps scaffold the problem-solving process, by helping students figure out the next steps of a problem. These sketches, then, reveal a student's mental model that can be assessed to understand the problem-solving strategy they used [111].

This dissertation illustrates how students and teachers use embodied representations in the form of metaphor, gesture, and sketching to make sense of computation and to support student conceptual learning. We propose that to better understand CS learning and quality CS teaching, further exploration of how body-based strategies are effectively is needed. However, embodiment is largely understudied within CER.

CHAPTER 4
RELATED WORK

In this chapter, I motivate why embodiment might support CS learning. There are a number of factors that contribute to making CS difficult. One of the most commonly reported is that CS has no physicality, and students need a way into the digital world of computation. In this dissertation, I describe how embodiment makes the abstract more tractable, which lends it to something students can use to "think with" and "think through." However, embodiment is understudied in CS learning. For example, research in CEd has considered metaphors to only have a minimal impact on learning or to only serve as a simple, teaching tool.

Besides reviewing the literature in CER, this chapter's goal is to highlight how little is known about embodiment in CER, which also motivates my methodological approach and basic research questions.

4.1 Complexities of Learning to Program

According to CEd researcher Sorva,

> "sometimes a novice programmer 'doesn't get' a concept, or 'gets it wrong' in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs. Unfortunately, misconceptions of even the most fundamental programming concepts, which are trivial to experts, are commonplace among novices and challenging to overcome" [64].

Programming is a fundamental activity in computer science. However, learning to program is hard. Concepts in computing are "precisely defined" and implemented. Students

need to reach ”particular ways” of understanding what constructs and concepts do, e.g., the flow of execution, variable assignment, what a variable is [64, 112]. This is complicated by, as Arawjo argues, programming languages are cultural tools with complex syntaxes and were designed and created in basements for professional use and not to support learning [113].

The difficulties associated with learning to program are well documented. Du Boulay argues that students find the concepts of programming too hard to grasp, do not understand the key properties of their program, and do not know how to control them by writing code [114]. In this thesis, I focus on one of the most common and difficult challenges to overcome: students having to deal with hidden, internal changes of the computer.

Computer programs have both static and dynamic forms. Sorva argues that the static form is visible in code, but the dynamic form is hidden [64, 112]. Nearly all misconceptions are results of aspects of constructs that are not readily visible, but hidden within the execution time [64]. Sorva defines a “misconception” by lumping together the concepts “misconceptions,” “partial understandings,” “student-constructed rules,” “difficulties,” “mistakes,” and “bugs” [64].

In a survey about the difficulty of programming concepts, Milne and Rowe found that concepts like pointers and memory were difficult because “these concepts are only hard because of the student’s inability to comprehend what is happening to their program in memory, as they are incapable of creating a clear mental model of its execution” [59]. It is reasons like this that consistently makes recursion one of the most difficult topics to learn.

Consider this example of a non-tail end recursive function (see Figure 4.1). Students have to be able to make sense of the behavior (how the code works) and function (the goal of the code) of recursive code, but the hidden, internal changes makes that difficult. For example, consider the following code snippet, an example of simple non-tail end recursion.

In this example, after the recursive execution ends, the rest of “recursiveFunction” still has to execute. Students typically have a misconception that after the recursive execution

```java
public static void recursiveFunction (int n) {
    if (n == 0) {
         System.out.println(1);
    }
    else {
        recursiveFunction(n - 1);
        System.out.println(n - 1);
    }
}
```

Figure 4.1: Code for a non-tail end recursive function.

is done, then that function is done and does not have to execute the remaining lines. Overcoming this misconception requires an understanding of execution frames, program state, the order in which frames execute, and where the recursive process is both virtually and physically, which the code does not make apparent.

Colburn and Shute distinguishes the kind of abstraction required in computer science versus mathematics [58, 115]. Whereas in mathematics, abstraction is about information neglect - or ignoring unnecessary parts to solve a problem, in computer science abstraction is about information hiding. Programmers create lines of code to instruct a compiler, but those lines of code do not show the complex calculations and operations that are actually happening. Computer programmers use abstraction to hide the material machine behind increasingly complex layers of code, layers which become a stack of abstractions, the lower ones hidden by the more complex layers on top. Students need a "way in" into the digital world of computation, and teachers need to scaffold that "way in."

4.2 Embodiment and CS Learning

The relationship between learning CS and embodiment is not immediately obvious. In physics learning, embodiment can be a natural and intuitive approach for students to learn the subject matter [116]. Euler and colleagues describe how students can metaphorically role play in processes like pushing each other on carts [116]. They describe how embodied

learning allows students to, ”relate their bodily intuitions to objects in otherwise physically nonintuitive domains” [116].

There have been interventions like the MoveLab [117] and CS Unplugged [118] that exploit the relationship between body movements and computational thinking to motivate and engage systemically marginalized students to learn computational thinking. However, Manches et al. argue that these interventions have not explained why embodiment might support CS learning beyond motivation and engagement [80].

Dijkstra argues that computers represent ‘radical novelty,’ meaning computers are “too novel to be represented by analogy or anthropomorphization” [9]. For that reason, students should not rely on embodiment to make sense of computation. There is some truth to Dijkstra’s statement; CS is different. Colburn and Shute describe computer science as a peculiar discipline concerned with creating its own subject matter [115]. According to Leron and Zazkis, mathematicians and computer scientists talk about a recursive process as progressing in different directions [119]. A mathematician takes an ”upward” approach or considers ”the base case” as the start point. They compute the factorial of a number as 1!, then 2!, then 3!, etc. A computer scientist takes a ”downward” approach or considers ”the base case” as the ending point. They compute the factorial using the knowledge they know (n-1)!, which can help them compute (n-2)!, until they reach 1!.

However, this novelty furthers the case for embodiment, or needing ways to make computing tractable and physical. Consider Papert, who argued that body-syntonicity, or using the knowledge and sense of one’s body, contributed to learning Logo because it helped students make the abstract concrete [120]. Papert noticed children identified with the Turtle they were programming by bringing their “knowledge about their bodies and how they move into the work of learning formal geometry” [120]. What Papert argues suggests that embodiment for learning CS is not only a problem-solving strategy, but grounds the abstract, and forms a representation that students can then think with and think through.

In this dissertation, I describe how embodiment helps students make sense of abstrac-

tions, by giving them something physical to think with and think through. Important to my argument, is that students and teachers rely on metaphors of physicality and space to describe and conceptualize abstractions. Moreover, I describe how teachers' use embodiment to explicate computation; however, they're use of embodiment may lead to confusion.

Part of the reason why embodiment has not been considered is because learning is typically framed from a cognitive perspective, as an individual act. In this research, I consider learning in CS to likely involves communication, negotiation, and interpretation.

4.3 Metaphors

Our study of metaphor in CS classrooms is different from recent research on the role of language in CS learning. Current CER typically investigates the ways programming syntax (e.g. for and while) creates barriers or misconceptions for both native and non-native English speakers [121, 122]. In my work, I do not consider programming syntax. Both my work and work on translanguaging are interrelated and collectively can provide powerful frameworks that support student learning.

CER typically considers metaphors and analogies as synonymous. Therefore, in this section, I also discuss the literature on analogies and CS learning.

Most of the studies dealing with the relevance of metaphors and analogies in CEd conceptualise metaphors and analogies as tools for teaching CS or have considered them unimportant. Some of this research has interviewed teachers to explain the kinds of metaphors they use to explain specific concepts. For example, Sanford and colleagues interviewed CS teachers about the kinds of metaphors they use in classrooms to explain programming concepts [8]. Although their findings are likely analogies, they produced a laundry list of metaphors, including describing functions as verbs or pointers as zombies. Given that metaphors eventually "break down," they asked teachers to assess "how far they could push their metaphors." They conclude that metaphors serve as good teaching tools, but that teachers are not aware of the limits of their constructed metaphors.

Other research has designed specific metaphors to teach computing concepts. Perez-Marín and colleagues used a methodology called MECOPROG using metaphors, such as, recipe/program, pantry/memory, and boxes/variables, to teach computational thinking [123]. Following an empirical experiment, they found that coupling the use of metaphors with a block-based programming environment (e.g., Scratch) has the potential to improve computational thinking knowledge acquisition in primary education.

Similar to my research, these avenues of research are also exploratory, however whereas my studies are naturalistic and observational, their work either designs a metaphor or asks teachers to describe the metaphors they use. Because my work is naturalistic, I found the kinds of impromptu, spurious metaphors teachers use.

Another strand of research in CEd does not think metaphors serve an actual purpose for learning to program or are primary culprits for students forming misconceptions [10, 124, 125]. Cao et al. conducted an experiment to understand if the analogies instructors use were effective [10]. They determined effectiveness by looking at whether analogies supported either near or far transfer using. They found small evidence that analogies were only useful for short-term learning and found no significant difference for long term learning and ability to transfer.

One of the reasons metaphors are chastised is because they "break down." Halasz and Moran describe that some features of an analogy may be useful for understanding computer concepts while other features may be irrelevant, which they argue adds to the difficulty of understanding concepts [124]. As Bettin argues this is a true statement, but not an argument against metaphor [63]. Bettin describes that when metaphor and analogy are grounded in elaboration activities and situated appropriately in context, we have already noted its effectiveness. In some ways, metaphors "breaking down" misses the point, because the metaphors still supports learning and it still helps students "gives them a foot in" to make sense of computation. Bettins' argument does a better job of treating learning as a progression.

Moreover, this work has a flawed understanding of how learning works. In Chapter 7, I studied how students actually learn and what they are actually doing, and metaphors served a great purpose. Metaphors provide a language to talk about abstractions, to help students reason and predict, and make sense.

Lastly, two papers have hypothesized that metaphors are likely how we conceptualize computing concepts.

Using Papert's notion of body-syntonicity, Watt argues that embodiment may be "important in the way we conceptualize certain computing concepts," communicating knowledge, and mediating learning to help students understand, comprehend, programming concepts [126]. Watt argues that we think about programs and programming languages as psychological entities: "After all, programs do things — they behave as if they have goals, the very components of Wellman's simple desire psychology" [126]. He concludes, then, that syntonic programming should make it easier for people to understand how and why programs do what they do, "because people can identify with those programs, and see things from their point of view" [126]. Important is that metaphors are "at the heart of the way that people grasp" programming concepts - and even programming languages, "conceptual metaphors might be the very mechanisms that we use to make sense of, reason about, and communicate computing concepts (similar to claims in mathematical cognition)" [126].

Building from this notion that metaphors underpin conceptualization of computing concepts, Manches and colleagues conducted a study where they analyzed the representational (or metaphoric) gestures college students produced while explaining CS1-level concepts (loops, algorithms, and conditional statements) [80].

In their structured interviews, they video recorded participants and asked them the following question: "Can you explain your understanding of loops/algorithms/conditional statements?" They found participants drew upon two overarching metaphors when explaining computation. The first was computing constructs as physical objects ("in which participants simulated manipulating physical objects (e.g., pinching) when referring to range

of computing constructs”). They also found gestures that “refer to a range of computing constructs including but not limited to: data, code, process, input, execution, output, conditions.” The second was computing processes as motion along a path, “whereby participants moved their hands along one of three bodybased axes when referring to temporal sequences.”

4.4 Gesture

Besides Manches and colleagues study on embodied metaphors in CS learning (described in the previous section) [80], my past study is the only other study that has sought to understand gesture production in CS learning [25].

Gestures, or spontaneous hand movements produced when talking, constitute a pervasive element of human communication and reflect human cognition [92]. While we often think of gestures that are used for emphases (e.g., head nods for the affirmative or head shakes for negative statements), they can also be visible, external representations of what people are thinking [127].

Gestures are an integral part of communication about concepts in the classroom. Teachers routinely gesture along with their speech. Gestures may play an important role in communicating knowledge to learners [101]. Students routinely gesture as they talk about the concepts they are learning. In a number of other academic disciplines, gestures have been identified as important aspects of understanding learning and improving learning.

For example, gestures produced during instruction and teacher-student interaction shed light on the mechanisms involved in learning from instruction [101]. Additionally, gestures externalize aspects of speakers’ knowledge, helping learners manage the ”working memory demands of mathematical thinking and explanation” [72]. In mathematics, understanding how gestures are used helps understand performance, instruction, assessment, and learning. Alibali et al. suggest that the study of gesture in mathematics helps explain why certain types of problems are more difficult than others, identify assessment methods that

accurately gauge knowledge, design more effective learning environments, select appropriate methods for instruction, and understand why learners have greater success with some instructional methods than with others [101].

Moreover, in learning sciences literature we find a number of analyses of people's interaction and conversation which include gesture [128, 129]. Some work uses gesture studies to provide insights to internal sense making [130], while other work seeks to incorporate multi-scalar analyses of gesture and movement to inform the production of social learning contexts and the design of learning experiences [131]. However, in computing education research there are few instances where the gestures of teachers or students are considered in analysis, but it is a promising direction for future work. Computing education might be able to draw on gesture research to provide similar insights to explain student behavior and provide insights into design of interventions.

We develop a conceptual framework to support future studies of learning and teaching that incorporate gesture studies in programming contexts. In particular, this paper introduces how gesture has been used to study teaching and learning, with a particular focus on one discipline (mathematics); critically reviews and interprets what concepts may be most relevant to programming contexts; and discusses what unique challenges programming contexts present to studies of gesture. We ground our explanation of the possible role of gestures by introducing examples from an observational study where we observed novice students learning to program. This paper concludes by suggesting potential avenues for future research in computing education that incorporate analyses of gesture in studies of teaching and learning.

Our contributions are from the lens of a scholarship of integration, where we consider how well the current gestures and learning literature integrates with issues in learning in computing. We offer an initial gesture taxonomy for computing education and suggest a research agenda to incorporate analyses of gesture in computing education.

4.4.1 A Gesture Taxonomy in Computing Education

While the previous work outlined above describes how gesture has been influential on understanding learning and teaching in mathematics there has been little or no focused work on describing how exploring gesture can help us understand computing education and learning. In the following sections we present an observational study of novices learning to program followed by an analysis of the type of gestures produced by teacher and students' interactions. Our aim in doing so is to make an initial exploration of gestures in computing education research and to set the stage for future studies.

Computing is characterized by the notional machine that cannot be perceived directly through the senses. Concrete and abstract have different conceptualizations in computing versus other abstract domains like mathematics. Mathematics typically has many lived experiences that one can draw upon to reason and understand. For instance, a gesture of a slope of a line can still be used through experiencing a slope of walking on a hill. However, in computing, concepts often do not have real-world, concrete counterparts. If one were to theoretically gesture a loop, what it represents in concrete is unknown. There is potentially a difference between pointing in computing versus in mathematics, and there may be gestures that have the same meaning and serve the same function. In the next sections, we ground the discussion of gestures in computing education and connect it the previous taxonomies through examples seen in a classroom observation study. We conclude with a discussion of the gaps in our knowledge about gesture in computing education.

4.4.2 An Exploratory Observational Study

We recently conducted an exploratory observational study of novice students learning to program in a classroom. The goal of this study was to gain an initial understanding of the type of gestures students and teachers produce in computing classrooms to help in building a taxonomy of computing gestures that could be used to facilitate learning.

During the Spring of 2017, the first author spent 90 minutes teaching an introductory

high school programming course once a week for 12-weeks. The curriculum for these 12-weeks included content on variables, if and if-else statements, while statements, and conditional statements. Class was in an alternative high-school which serves students who had challenges in traditional academic learning environments. There were eight students: six males and two females, seven Black students and one Hispanic student. Students had varying level of math skills and were taking either Algebra 1 or Geometry. None had experience with Calculus.

Each class period, students worked on different Scratch projects, ranging from music video creation to creating a game based on a social justice issue. We often had students work in groups of two or three and present their work for peer student critiques.

We taught them variables in a lecture. Students individually completed a worksheet about variable manipulation and declaration and then the class discussed the answer to each question on the worksheet. We taught conditionals, if-statements, and loops by having students convert their favorite lyrics to code. The students wrote the "code" on a white board and had to explain it to the class. We would also write "code" on a white board and would have students explain it.

At the end of the 12-weeks, we conducted 15-minute structured interviews with each student where we had them tell us the output of eight pieces of code. Each piece of code covered different computing topics (variables, if-statements, and while loops). We had students explain their responses since explanations tend to be a rich source of gesture data.

The first author took observation notes at the end of each class period. Observation notes were taken using a two-column approach, with observations in one column and corresponding reflections added after the session in another column. When students were observed making gestures, we wrote a description of a gesture and its accompanying speech. The observation and gesture notes were analyzed to identify types of gestures, how they might map to existing taxonomies of gestures, and the student knowledge they might convey.

In the Fall of 2017 we worked with the same students, conducting a 4-day programming workshop. This gave us an opportunity to observe certain gestures and ask for further explanations based upon questions from our first pass at analysis.

We are not presenting a study that tests a hypothesis or evaluates an intervention. Our data cannot be independently verified, and at best, we can say our analysis is a plausible interpretation. However, for an initial study, this level of observation and reporting is reasonable to provide foundational work in this under-researched space. We are noting what gestures we saw and connecting them to existing taxonomies. With more precise methodology and data collection (e.g., videotaping interactions), future studies could produce data that could be more rigorously analyzed.

4.4.3 The Types of Gesture in a Computing Classroom

In this section, we present the different types of gestures we observed in the computing classroom that comprised our study. We then describe how these gestures were used during instruction within this context and also what these gestures potentially revealed about student knowledge.

Deictic

As described previously, deictic gestures, or pointing gestures, point to establish the location of an object [108]. In mathematics, these gestures reflect the grounding of cognition in the physical environment [101].

In our study, deictic gestures may also reflect the grounding of cognition and may help uncover students' understanding of code execution. Computing teachers often face the challenging task of helping their students see connections between different ideas, events, or lessons. When explaining concepts to students, we would gesture by pointing to the lines of code we were referring. These gestures seemed to help guide students' attention to those inscriptions; for example, we indicated that a variable was being manipulated in different

spaces, causing the value of that variable to change by pointing to each line of code.

In mathematics, pointing to these inscriptions while providing explanations helped link the teacher's verbal stream to its referents and ground the cognition. This may also be the case in computing. Grounding makes information conveyed verbally more accessible to students, thus fostering students' learning of content and scaffolding students' understanding [132]. In computing, these deictic gestures seem to help show process: what the code is doing (or at least what the person thinks it is doing) during execution. Thus, when teachers use deictic gestures, it may help scaffold students understanding of the way code executes.

Students also used deictic gestures. During interviews, some students gestured while they tried to understand the control flow (see Figure Figure 4.3). Most students that did not gesture while trying to figure out the code were not able to correctly predict the output. These gestures may help students trace code, helping students reason about answers. These types of gesture may serve as a problem-solving function, helping to bring about strategies for reasoning. Examining student's gestures while they problem solve, may help reveal misconceptions and how well they understand code execution.

Square brackets indicate the words that accompany each gesture. Gestures are numbered below the bracketed corresponding speech and described in detail below the speech transcript.

Teacher: So, tell me what you think the output is

Student: Well, it will [check if this is the case] and since

1

it's not, it won't do this and [will go straight here]. I

2

think the cat will say "yes."

1. Right index finger points to first line of the if-statement

2. Right index finger then points to the line after the conditional statement

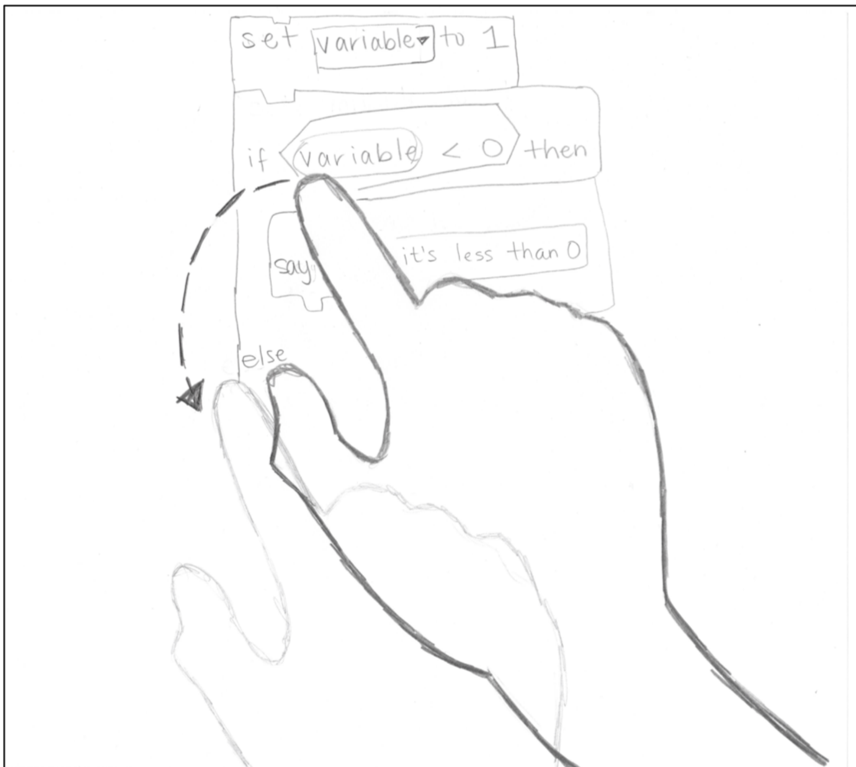


Figure 4.2: A student using deictic gestures while they trace a program's control flow.

Iconic

As we previously discussed, iconic gestures represent concrete ideas and are used to convey information about the size, shape, or orientation of the object of discourse [133]. In mathematics, iconic gestures manifest mental simulations of action and perception [101].

In computing, iconic gestures also seem to manifest simulations of action and perception and can help to understand the different conceptual models that students are forming. In mathematics, these gestures are produced when someone thinks and speaks about mathematical ideas. Alibali et al. suggest these gestures may be intentionally produced to facilitate thinking about such ideas or to communicate such ideas [101]. In computing, these gestures seemed to help facilitate communication and simulate actions.

Students would often have to present their work to the class. Before presentations, we would give a brief lecture on how to talk about code, telling them not to say what each line was doing, but to just generally say "my code does this, and I used this if statement, for

example, to do it." During presentations students often used iconic gestures to help them talk about their code.

Student 1: My code rotates the cat a certain number of times depending on where it is. My code first figures out where the cat is and depending on the location, [it keeps going until...]

1

Student 2: My code rotates the cat a certain number of times depending on where it is. To do this, [I have this variable, then I have an if statement that says if the cat is here...]

2

1. Made circular gestures with both hands to show understanding of repetition.
2. Used right index finger to point to each line.

In that example, we noticed students who did not use iconic gestures, and who instead used deictic gestures, talked about their code at a low, code-focused level. Chu and Kita describe the shift from pointing gestures to iconic gestures shows a change in an internalization of action strategy, which had become less tied to concrete actions and thus more abstract [134]. Students' gestures might match how abstracted their knowledge is. Understanding how gestures transform over time, could have implications for formative assessment.

Metaphoric

Metaphoric gestures, as illustrated earlier, are similar to iconic gestures; however, they represent an abstract, and not concrete, idea [92]. In mathematics, metaphoric gestures reflect conceptual metaphors that underlie mathematical concepts [101, 135].

In computing metaphoric gestures might help communicate abstract concepts and might uncover misconceptions a student has about a concept that could be difficult to understand or even notice through other means.

Computing instructors have the challenge of helping novice students build strategies and mental models. Like in mathematics classrooms, computing teachers often use various teaching methods to do this, including using metaphors to teach abstract concepts [136]. Metaphors provide a strategy for understanding, by mapping abstract concepts to familiar, real-world concepts [8].

We often used metaphoric gestures when describing concepts to our students. During the lecture on variables, we described it as "some type of object that holds something." We used a gesture forming our hand into a cup, suggesting the metaphor of a variable as a cup, capable of holding something.

One feature about metaphors is they have limitations: "A feature of many metaphors is that when their mappings are pushed beyond their user's intended limits, they eventually break down" [8]. The metaphors used to describe computing concepts might lead to misconceptions about computing concepts. There have been catalogs of misconceptions students have about computer behavior and basic programming constructs [137]. One of the examples of a misconception alludes to the conceptual metaphor students use to understand the concept: "students commonly consider classes to be containers for objects" [137]. Teachers use of metaphoric gesture might communicate unintended misconceptions to students.

Teacher to the class: Does anybody know what a variable is?
Student: Uhhh, I guess something that changes.
T: Well, yeah that's what variable means, and I guess that relates, but what about a variable in programming?
Class: blank stares
T: A variable is some [type of object that holds something].

1

It's like a container. If there's some value or data you want to store or keep for later, you put it in a variable.

1. Right hand forms a cup, while left hand goes inside the ”cup.”

Students also used metaphoric gestures when describing their code. A student was having trouble building a game. We noticed the student was using a loop and asked the student to explain that code segment (see Figure Figure 4.3). They described a loop as “a loop,” similar to a roller coaster loop.

It is helpful for instructors to know the misconceptions of students in order to provide appropriate help. Metaphoric gestures may reveal the underlying conceptual metaphors students have which can reveal uncovered misconceptions. This may also help reveal where their misconceptions come from if they have a conceptual metaphor that does not map well to the computing term.

In the example (Figure Figure 4.3), although their definition of loop seems correct, the students metaphorical gesture potentially reveals a misconception they have about loops. Thinking of a loop as a roller coaster loop is correct in that both types of loops have a start and end point. The student, however, might not completely understand a loop conceptually. Their gesture moving backwards (or from right to left) could suggest a misconception of state changes.

Teacher: Wait, tell me about this. What is this doing?
Student: Oh that’s for if they still have lives they can keep playing the game. [It’s a loop, so it’s like a loop, it repeats].

1

1. Right index finger moves in several loop-de-loops, going from right to left.

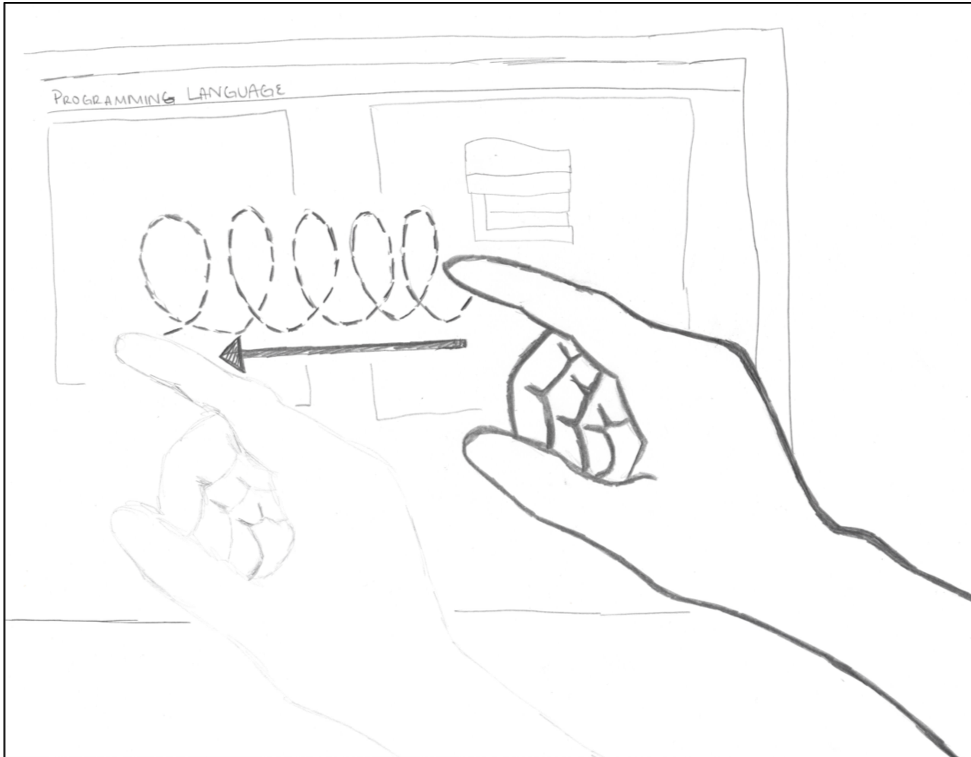


Figure 4.3: A student using a metaphoric gesture while they described a loop.

Beat

As discussed previously, beat gestures are the simplest type of gestures - a simple "kinetic realization of the underlying pulse" [133]. Beat gestures do not depict specific content but are just gestures timed to important content words. Beat gestures are talked about in mathematics, but do not convey semantic content and thus have no meaning.

We did not see beat gestures in our observations. However, we can *imagine* what a beat gesture might mean in computing education.

In computing, beat gestures could be used to represent iterative process across a sequence of data. Novice computing students have difficulties understanding memory-related concepts because they are abstract [59, 114, 138]. Teachers could use these gestures as a visualization for showing data or memory in some space. A teacher might produce beat gestures when they talk about iterating through an array or other collection. While the teacher talks about iterating through each index, the teacher may "point" to each index in

space. The pace of the gesture may indicate the iteration where a function or body of the loop is being applied to each element of the data.

4.4.4 Discussion

We were challenged to fit into McNeil's taxonomy the gestures that computing teachers and students produced in our study. There were obvious instances where the taxonomy fit. For instance, when people were pointing to text, the gesture was obviously deictic. Many of these gestures could technically fit into different types. The mapping of McNeil's taxonomy is not one to one with computing.

We see two particular challenges to developing a gesture framework for computing education. First, deictic gestures ground cognition by connecting thought to real world objects. While a deictic gesture does direct attention and ”place (cognition) in the real world,” it is hard to say what grounding means when pointing in computing. In math, if a teacher points to a 3, students have multiple sense of the number “3” to use in grounding to the real world, e.g., any set with three elements, the numeral 3, the word “three,” a picture of three things, and so on. Wilensky has argued that abstraction to concreteness is a spectrum, and concreteness reflects the number of ways in which someone can reference the concept [139]. In computing, if a teacher points to a variable or a line of code, there is no physical or real-world counterpart to help that make sense. If the point of deictic gesture is to connect the abstract to the concrete, we in computing have very little concrete to reference. There are lines of code on the screen, and there are behaviors in the world.

Second, what we categorized as a beat gesture could also be a deictic gesture. If we are pointing to data (even a spatial, gestural sense of data), we are technically pointing to *something* in a location. There are gestures that appear in computing education that do not fit cleanly into the taxonomy.

We suggest that the complexity arises because we in computing are referencing both process and data. The deictic gestures that we observed seemed to more often refer to

process. The producer of the gesture most often was describing the way code flows or executes. Beat gesture may be more commonly used to reference data elements, but at a pace that represents process.

We were similarly challenged to distinguish between metaphoric gesture and other gesture types like iconic and beat. We had to loosen our definition of concrete versus abstract. Instead, we focused on what information was being conveyed. There is a duality of computing because computer programs can be characterized by their physical implementations on physical devices (i.e., there is a physical device that is running the program) and their conceptual implementations in programmers' minds which does not necessarily know or care where the physical device is located [140]. Programmers' create conceptual implementations metaphors, e.g., like imagining a "server farm" or "cloud" where the program is being executed. Many of the gestures produced while explaining concepts, could in fact be metaphoric gestures. But the metaphors may not lend themselves to mapping to gestures, e.g, the challenge of pointing at "the cloud."

By including gesture in our analysis of computing education, we believe that we add a new perspective to our research, and we may be able to expand the roles that gestures play. Computing is different than other disciplines because of how we play with concrete and abstract, with process and data. Gestures are important to cognition, so they can help us gain new insights into cognition and learning. But the role of gesture may be different in computing than in mathematics (or other STEM disciplines), which gives us a unique lens on gesture. The study of gesture in computing education gives us an opportunity to advance our own knowledge and that of the learning sciences, too.

4.4.5 Conclusion

In this research, we are starting to explore gesture in computing education. Our goal is to identify connections to relevant literature on gesture, raise important questions for computing education, and generate interesting hypotheses for future testing. From our analysis, we

saw gesture potentially used as a problem-solving strategy, as a way to communicate one's understanding, and a way to communicate abstract concepts. Devlin suggests computing is about constructing, manipulating, and reasoning about abstractions [141]. Studying gesture in computing could help us understand how students reason about abstract ideas and help us understand the different strategies students use to make sense of computing. By studying gesture, we might develop better ways of presenting and communicating knowledge to novices and better understand how novices are communicating their misconceptions and understandings to teachers. Likewise, the study of gesture could lead to new instructional practices that lead to more effective learning.

4.5 Sketching

Findings in CER have argued that sketching a code trace leads to greater success. Sorva defines tracing as, 'analyzing its execution to determine what operations occur and how its state changes" [142]. This research has argued that sketching leads to success because it both offloads and distributes cognition and makes some information more salient.

Research by Fincher and colleagues [143] and Cunningham et al. [21] reveal that some students often draw pictures, sketch code traces, or perform calculations when programming or solving programming-related problems. Holliday and Luginbuhl conducted a study to evaluate how well having students construct diagrams about the execution of object-oriented execution can serve as an assessment tool [144]. They found a correlation between "a student's ability to draw visual representations of objects in a program's heap and their comprehension of the material". They concluded that students that can diagram what is happening in memory suggests that they can more easily and deeply understand the meaning of the program.

Likely, one of the most influential studies on sketches in CER was conducted by the Leeds Working Group [145]. They conducted a multi-institutional, multinational study where they categorized and analyzed the drawings and sketches students created on test

sheets. They created a taxonomy of sketches, where they describe which sketches are likely more helpful for answering a problem correctly.

Cunningham et al. replicated the Leeds Working Group study and created a categorization of the sketches students make when tracing code [21]. They found that students who did not sketch did not perform as well, and concluded that sketching is “a technique to distribute cognition and manage cognitive load” in CS. Interestingly, she suggests that “students who have a good understanding of the notional machine are more likely to sketch than students who don’t.”

Cunningham et al. defined a sketch in computing as the following: ”a programmer’s written visualizations of program state or any other computing process” [21]. Sketches, and by extension, sketching, is typically considered a mechanism to manage cognitive load. However, when viewed as a body-based activity, sketching is an effective activity for reasoning because it acts as ”an external mediating structure” [109]. The act of creating the externalization primes someone’s thinking, and the ”aspects” a person chooses to sketch highlights their subject of reasoning.

In this dissertation, I focus on sketching, that is, I focus on the process of drawing or creating a code trace, and not on the final product, unlike past studies in CER. While we typically view sketching as just off-loading cognition, my work has us rethink the process of sketching, and not the final product. By focusing on sketching, I analzye as a way to embody process. This shows a better way of how sketching primes students to think about “what comes next” “what happens next” “where does the code go”.

CHAPTER 5

EMBODIED REPRESENTATIONS IN COMPUTING EDUCATION: HOW GESTURE, METAPHOR, AND SKETCHING SUPPORT TEACHING RECURSION

5.1 Introduction

I conducted a grounded analysis of a set of naturalistic video recordings of university professors teaching recursion in their classrooms. I wanted to understand how teachers use space in CS classrooms. Therefore, I looked for spatial representations, including iconic gestures, spatial language and spatial metaphors, and artifacts, such as sketches or diagrams. While conducting this analysis, I noticed an interesting relationship between space and embodiment. The gestures the teachers created about computation seemed to be enacted metaphors; that is, the gestures seemed to act out a metaphor. Similarly, the teachers used spatial language (e.g., then, it goes down here; then, I jump up to the if-statement) while describing a code trace and used interesting language, such as calls, runs, and bound, to refer to function invocation, program execution, and variable assignment, respectively. However, all these are metaphors. The computer is not going down or calling anything, but teachers used metaphors and physicality to represent the abstract. Moreover, teachers metaphorically construe themselves as executing lines of code or constructing an agent who does that. Embodiment seems to play a central role in the ways teachers communicate information and meaning. This study sets forth the conceptual framework of the kinds of embodiment used to analyze the rest of the work.

A version of this chapter was published in the International Conference on Learning Sciences (ICLS) 2020.

5.1.1 Motivation

Understanding how teachers teach is essential to understand how students learn. Peter Burton highlighted this when he suggested computing education research (CER) should be critical of the modalities between "what actually gets taught; what we think is getting taught; what we feel we'd like to teach; what would actually make a difference" [146]. Recently, other computing education researchers have expanded this view, contending that CER needs to move beyond individualistic theories of cognition to explain teaching and learning [39]. Otherwise, we miss important details that describe how teachers orchestrate available resources to scaffold students' understanding and what "knowledge students utilize to make sense of the problem-solving activities in computer programming" [39].

In this chapter, we critically reflected on the current practice of instruction, using embodiment as a lens to analyze teaching practices. To understand how the teaching of computing is embodied is to know how embodied representations support the reasoning involved when thinking computationally and expressing computational ideas. Evidence tells us that embodied representations (i.e., gesture, embodied language like perspective-taking and conceptual metaphors, and tool use) are pervasive in computing classrooms, which suggests that embodiment may be central to both the learning and teaching of computing [21, 25]. However, in CER, we have little discussion about how instructors use their bodies when teaching the skills involved in "doing and learning" computing. Learning and the practice of computing are not "purely intellectual activities, isolated from social, cultural, and contextual factors" [39], but are dependent on our bodily-grounded experiences and opportunities.

We investigated the following research question: What embodied representations do teachers use while teaching in computing classrooms? Answering that question is the first step towards understanding how embodied representations can support students' reasoning and learning in computing classrooms. To answer the question, we present two case studies where we studied the teaching of recursion in computing classrooms to understand how

teaching was embodied during instruction. We selected recursion because the difficulties with learning the topic are well documented in computing education literature, and it is a topic that is generally agreed upon that every computing student should know [55]. Drawing from contemporary theories of embodied cognition [6, 87, 109], we used grounded theory to analyze a set of naturalistic video recordings of undergraduate computing professors teaching recursion to their class. We paid particular attention to how the professor used embodied representations - i.e., gesture, embodied language, and tool use - to support the learning of recursion. We contribute a conceptual framework of the types of embodied representations teachers use in computing education, which we elicited from the two case studies. The paper concludes by suggesting how these uses of embodiment in teaching recursion may impact student learning.

5.2 Background

In general, embodiment implies that the mind is not the sole source of knowledge, but we make meaning about the world from our body-based, lived experiences [6]. Using embodiment in the analysis of teaching practices, therefore, shifts the unit of analysis from the individual's mind, to an activity that is culturally and historically situated. Teachers are using embodiment to express computation, even if not intentionally (e.g., through gestures). Teachers cannot help but use embodiment because they are themselves embodied, and students are watching them. Students are sensing the motion and gesture, even if teachers are not intentionally using gesture and other embodiment. Consider Seymour Papert, who argued that body-syntonicity, or using the knowledge and sense of one's body, contributed to learning Logo because it helped students make the abstract concrete [120]. Moreover, interventions like the MoveLab and CS Unplugged exploit the relationship between body movements and computational thinking to motivate and engage underrepresented students to learn computational thinking [117, 118]. However, CER has yet to view classroom instruction of computing as an embodied activity. As a consequence, computing education

theory is missing a significant part of how educators support student reasoning and learning (even if the educators are unaware of it consciously). We need exploratory, ground-up research to develop theory to think about the relationship between teaching and embodiment, and its impacts on learning. We contribute a conceptual framework to understand the embodiment in computing instruction. We focus on three embodied representations: gesture, embodied language, and tool use. These representations were the most salient in computing classrooms and have practical implications for pedagogy.

5.3 Method

As previously mentioned, I present my findings in the style of detailed episodes; this work is meant to be descriptive and interpretive. I provide my findings as episodes not only to present evidence to readers but also to enable them to come to their own conclusions, which might be contradictory to the ones I make. Moreover, as this work is interpretative, I do not provide exhaustive lists or categorizations.

5.3.1 Data Sources

The video data that was analyzed in this study was collected as part of an exploratory study about how spatial representations (i.e., gesture, spatial language, and visualizations) appear in computing teaching in classrooms. In total, we gathered 227 minutes of video data from four undergraduate professors. We emailed professors at universities that are allowed to video record their classrooms, scoured MOOCs, and online video databases to create our video corpus. We received or looked at 33 videos. Because the study was based upon who responded and who was able to send us video recordings, it potentially provides a skewed view of what happens in computing classrooms. This is a limitation for two reasons. First, all professors were middle-aged white men at universities in the United States of America. Second, professors in the videos are known as excellent computer science teachers. However, this sample is representative of teachers and teaching practices that are rewarded by

institutions – these professors were all tenured at prestigious computer science departments – and modeled by others. Each course was taught in a different imperative language. All the courses were ”conventionally structured” [50], with Socratic-style lectures in an auditorium classroom and practical laboratory work in another session. The videos show only the professor, but some student voices are intelligible. For this paper, we review two of the instructional moments related to recursion from two instructors. These two were selected because they were the most salient examples of embodied representations. The two cases offer us concrete examples in which to talk about a conceptual framework that can be used to think about embodiment in computing education. Our goal is not to make any confirmatory or nonconfirmatory claims about intentionality or if the students understood the embodied representations but to document the use of and to provide an initial conceptual framework of the embodied representations used in computing teaching.

5.3.2 Data Analysis

The video data were analyzed using grounded theory [46]. For each recording, we first reviewed and transcribed the entire recording to get familiar with the content of the video, and then created a timestamped content log for each recording. A micro ethnographic approach to data reduction was used to ’tag’ moments that we believed indicated the teacher’s use of embodied representational content. We identified those moments from our content logs. We then created multimodal transcripts of the moments produced in style inspired by Goodwin [147]. These tagged moments were first analyzed inductively to form our theoretical understanding of the ways teaching was embodied during instruction. Afterward, we reanalyzed the codes and classified them deductively as iconic, noniconic, or deictic gestures, embodied language, or tool use. Two researchers played back the videos within video analysis software repeatedly with and without sound to pay attention to when and what type of gesture was produced. Deictic was any pointing gesture, iconic was any gesture with a concrete referent, and noniconic, to refer to gestures that could be metaphoric, or repre-

sent abstractions and computation, or could be nonsense or a communicative gesture, like a thumbs up or okay sign. We studied the transcripts to identify utterances containing ontological metaphors and perspective-taking, i.e. when the teacher used first or third person. Tool use was anytime a professor created a sketch of a code trace. Note, only one professor in our corpus created a sketch. For an in-depth data analysis, for each representation please see Chapter 6.

5.4 Case Studies

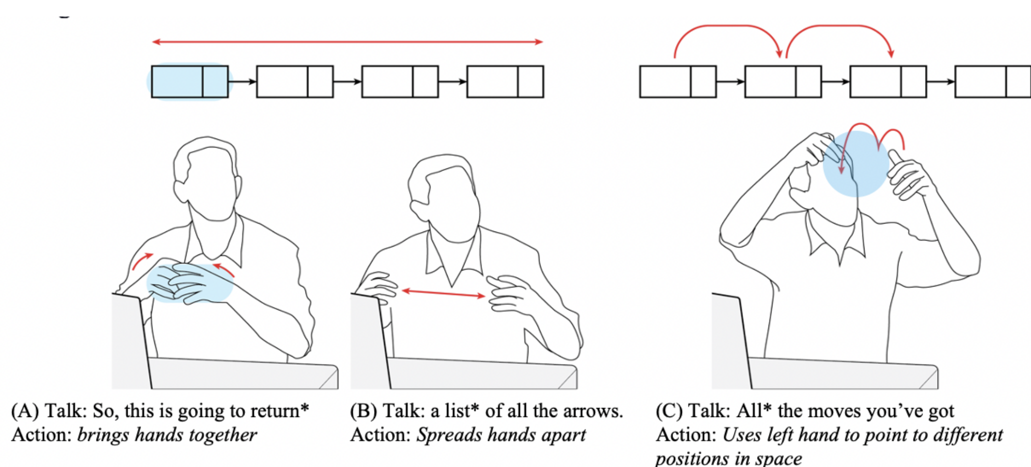
In this section, we present two case studies in the style of vignettes to illustrate how gesture, embodied language, and tools were used by two computing instructors.

5.4.1 Case Study 1: Gesture

This series of excerpts highlight the ways the professor used gestures and other embodied representations, while collaboratively writing a recursive program. The class is a more advanced introductory course and taught in Scratch. There was a table in the classroom that the teacher sat at with their computer connected to a projector. During the 23-minute video excerpt analyzed for this case study, the professor is reviewing recursion for the class's upcoming midterm. The professor has the class participate in a "group programming session," where the professor, in real-time, writes the code for a game on the computer, which is then projected for students to see. The professor asks the students to help him figure out the code. While describing the rules of the game, the professor projects a diagram of a tree (see Figure 5.2.A) for students to see and uses a hand gesture that resembles the tree diagram. A tree is a type of data structure, and it is a standard convention to state that a tree has "children" nodes. The professor tells the students that they have to write three functions: (1) a function that generates a new position, (2) a function that returns a value from a list, and (3) a recursive function that iterates through the list. Then, the professor takes down the diagram of the tree, opens the Scratch programming environment,

and begins writing code.

The professor quickly writes the first function since it only had one line of code. To write the second function, the professor asks the students how "they might get some value from the list." Figure 1 is the transcript of a clip of this interaction, which shows the instructor using a noniconic gesture that represents a list (a type of data structure, see Figure 5.1.A and Figure 5.1.B) and a series of deictic gestures to provide a visual of each element in the list as if he is iterating through the list (see Figure 5.1.C). The professor "points" to different locations in space as if he were pointing to different chunks of data in the list. The intention of that gesture seems to be to help students reason about the logic for the program. The noniconic and deictic gestures seem to provide the instructor's mental model of a list, which could help students conceptually understand the functionality of a list. Consequently, the gestures appear to provide a concrete example that a student could use to reason about how the function might work. Frequently, conceptual models of lists show a linear array of "chained" boxes (see Figure 5.1), and his movements seemed to mimic this conceptualization.



(A) Talk: So, this is going to return*
Action: *brings hands together*

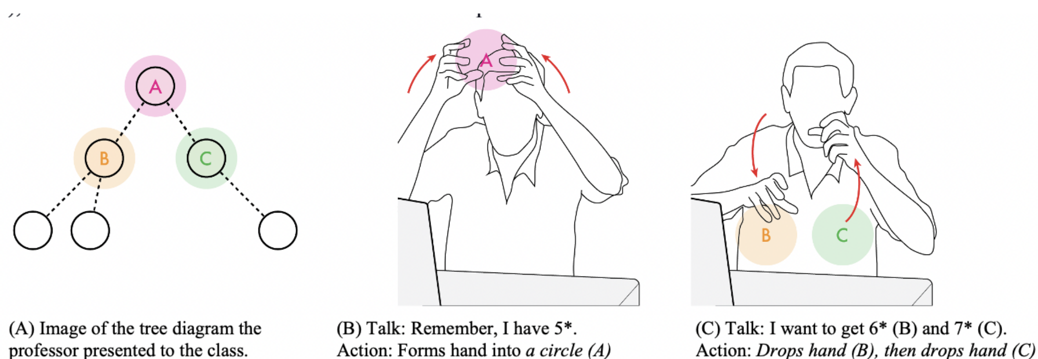
(B) Talk: a list* of all the arrows.
Action: *Spreads hands apart*

(C) Talk: All* the moves you've got
Action: *Uses left hand to point to different positions in space*

Figure 5.1: The professor uses metaphoric and deictic gesture to describe a list. At the top of this figure is a typical conceptualization of a list. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image.

After writing the code for the second function, the professor describes the logic for the recursive function. First, the professor describes the base case. While repeating the series

of deictic gestures of iterating through a list, he asked the students, “When are you done? If not at a leaf node, end of list, then these things happen.” These gestures could connect “being done,” the base case, with the list reaching a specific state, which provides a visual representation that could help students understand when the recursive invocation ends. Next, the professor describes the recursive case, saying, “they need to understand what children the tree has.” Figure 5.2 is the shortened transcript of this interaction. The professor used the same noniconic gesture that resembled the tree diagram – without projecting the tree diagram – by raising his arms and placing his hands in a circle as if showing a specific node’s placement in the tree. He then moves his left hand, followed by his right hand, as if he were traversing a tree to get to another node (see Figure 5.2.B and Figure 5.2.C). Repeating the series of gestures and gesturing “traversal” appears to help students recall the knowledge they need to solve the problem and could help with learning recursion by making visible the recursive function execution.



(A) Image of the tree diagram the professor presented to the class.

(B) Talk: Remember, I have 5*.
Action: Forms hand into *a circle (A)*

(C) Talk: I want to get 6* (B) and 7* (C).
Action: *Drops hand (B), then drops hand (C)*

Figure 5.2: The professor uses metaphoric gesture while describing the tree and children nodes. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image.

5.4.2 Case Study 2: Metaphors and Sketching

This sequence of excerpts highlights an instructor’s use of embodied language and tools when defining recursion and tracing a recursive function. The class is an advanced introductory course and taught in C++. The teacher had a podium with a computer and stylus, and a connected projector. During this 17-minute video clip, the professor is introducing

recursion to the students. First, the professor describes recursion with an analogy of looking up a set of words in a dictionary in combination with a visual-gestural narrative that acts out the analogy, seen in Figure 5.3. The professor points to the palm of his hand as if he was reading from the dictionary (see Figure 5.3.A), then uses a sweeping gesture to indicate looking up the definition of other words that were part of the definition of the first word (see Figure 5.3.B). Then, the professor describes recursive solutions with the ontological metaphors "powerful" and "elegant:"

The contrasting metaphors invoke feelings of strength, influence, and daintiness. Most recursive solutions are considered "good code," since they have considerably fewer lines of code than solutions that use for-loops yet are functionally the same. The embodied representations and analogy appear to invite the students to use their lived experiences to understand, functionally, how recursion operates, and when recursion is appropriate to use in a code solution.

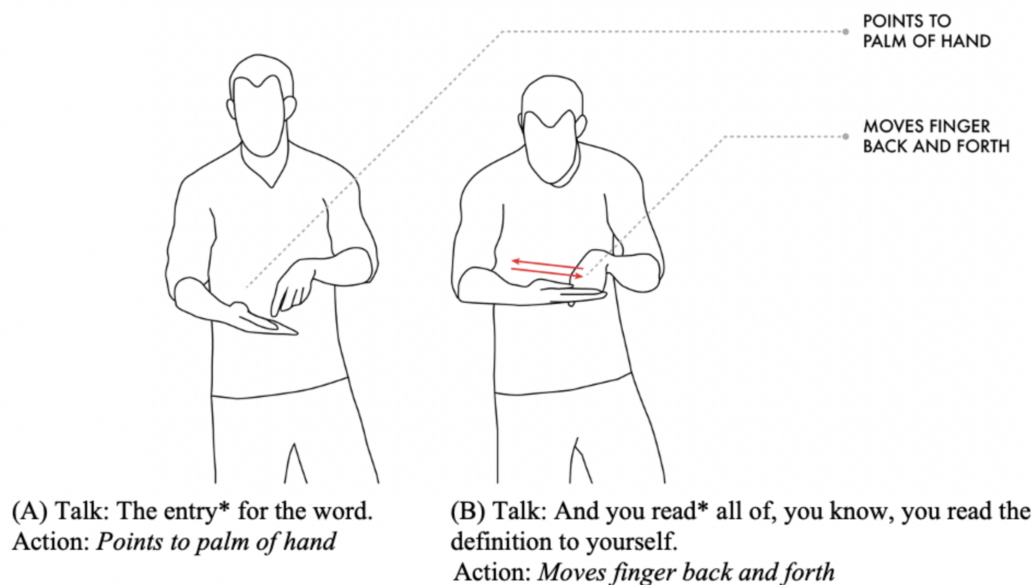


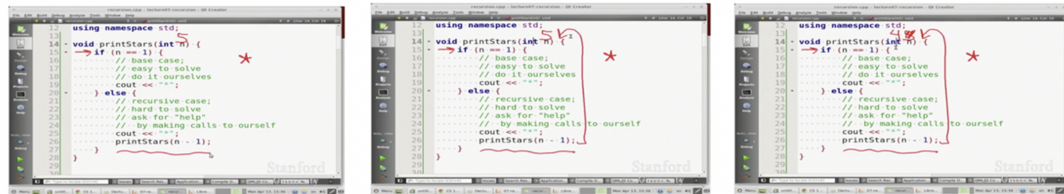
Figure 5.3: The professor uses metaphoric gesture while describing the tree and children nodes. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image.

Next, the professor has the class participate in a group programming activity to rewrite a function to use a recursive solution instead of a looping solution. The function, printStars,

produces ‘n’ number of stars. After writing the solution, the instructor sketches a trace of the recursive execution. The sketch of the code trace allows someone to “see” the process of a recursive execution. Figure 5.4 is a segment from the interaction. The professor writes a “5” above the statement (int n) to show the students that the parameter ‘n’ has a starting value of 5. Then, he draws an arrow next to the statement if (n == 1) to show the students that the program will begin by evaluating that condition. After determining the condition is false, he draws the output – a star. He then underlines the recursive invocation, emphasizing the function invoking itself (Figure 5.4.A). Notice the professor uses embodied language: “Then it says “... “now I have to call.” The statement, “then it says,” suggests the professor is using perspective-taking, by stepping out and giving the recursive invocation “agency.” Then, the professor switches perspective, “now I have to call,” as if he is diving into the code, experiencing the invocation. The professor also uses a common type of ontological metaphor, personification, when he states that “he” has to “call” the recursive invocation, similar to calling someone over the phone. Next, the professor draws an arrow that points back to the function while saying, “so it makes it call this function again” (see Figure 5.4.B). Again, the professor uses personification and perspective-taking, but this time to “show” a function invoking itself. Then, the professor crosses out the ‘5’ and draws a ‘4’ (see Figure 5.4.C). Cunningham et al. refer to this sketching technique as the crossout method. They suggest that it “demonstrates that previous values are no longer accessible by a strike-through” [21], which is an excellent way to depict to students a function reusing a parameter: a concept students typically have trouble understanding when learning recursion.

5.5 Discussion and Conclusion

We described two case studies to show how computing instructors use embodied representations - in the form of gestures, embodied language, and tool use - to teach recursion. Table 7.1 is a summary of the embodied representations found. We contribute a conceptual



(A) Talk: Then it± says, “now I¢ have to call° printStars with n minus 1.” n minus 1. 5 minus 1. That’s 4.
Action: *Underlines printStars(n-1)*

(B) Talk: So that makes it± call° this function again.
Action: *Draws arrow pointing to the function header*

(C) Talk: But, with an n of 4, right?
Action: *Crosses out the 5 and draws a 4*

Figure 5.4: The professor sketching a code trace while using embodied language. The professor’s talk indicating embodied language are noted: stepping out±, diving in¢, ontological metaphor°

framework of the kinds of embodied representations teachers use in computing classrooms as the first step towards understanding how embodiment supports student learning. Next, we hypothesize two questions that are important for future research to consider: (1) Which of these embodied representations are intentional, and which are not? (2) What is the effect on student learning?

Table 5.1: A summary of the embodied representations teachers used.

Embodied Representation	**Example**
Gesture	
Deictic	Case Study 1: Professor "pointing", suggesting iteration
Metaphoric	Case Study 1: Professor gestures a "tree"
Embodied Language	
Ontological Metaphor	Case Study 2: When the professor said, "that makes it call"
Perspective-Taking	Case Study 2: When the professor said, "Then it says"
Tool Use	Case Study 2: Professor sketches a code trace

The goal of a computing instructor is to help students form viable mental models of how the computer works. However, to form viable mental models, learners must understand abstract concepts that cannot be perceived directly through the senses. Herb Simon argued that computing is a science of the artificial, something ”designed” and not inherited from nature [14]. For example, the discipline relies on metaphors based on lived experiences, e.g., naming procedures or subroutines a function, to name the abstractions [115]. Computing education researcher Ben Shapiro argues that because computing is a science

of the artificial, we cannot understand learning about computing without interrogating the practices and sociocultural contexts that help students with understanding and reasoning [20]. Our findings indicate why it is important to use embodiment as a lens to understand learning in computing education: the embodied representations are likely some of the few resources students can use to understand the abstract. We know that students attend to ”shallow features.” Movement, gesture, language choice, and tool use are features we might expect a novice to attend to because they do not know enough about what is not important to attend to.

If we assume that students are attending to these embodied representations, then they likely affect student learning. We discuss two ways the embodied representations possibly supported student learning. In case study 2, we identified instances of embodied representations in the professor’s language and tool use when he sketched a code trace. Research about program comprehension indicates that it is hard for students to form mental models of code execution because they do not understand statement sequencing [50]. Therefore, one interpretation is that the embodied representations supported students’ formation of viable mental models by ”concretizing” statement sequencing and letting students ”symbolically live the experiences” of function invocation. Sketches, then, could have helped concretize statement sequencing in two ways. First, the teacher’s sketch of a code trace showed his mental model of recursive invocation by drawing the critical ”aspects” that students need to know to understand statement sequencing [109]. Second, the act of sketching could prime students to make predictions and think about what happens next. Research in science education shows that students often find it helpful to identify with individual elements in a model, and then view phenomena from the perspective of this element [87]. Using embodied language likely ”immersed” students into the function invocation - as if they were physically in it - helping them envision statement sequencing.

In Case Study 1, we described an instance of the professor using both deictic and noniconic gestures to describe the functionality of a list and to ’act out’ iterating through a list.

The professor likely used embodiment to make salient the critical aspects of a list, i.e., iteration, which provided a utility for and the knowledge needed for students to reason about the logic. Moreover, noniconic gestures typically depict concepts that are challenging to describe in words and are ”shaped” by particular objects. Scopelitis et al. argue that, “the gesturing hands can be employed as tools to build a representational object that both the speaker and the hearer act upon in order to achieve a shared understanding of a complex concept” [148]. Therefore, when the teacher used noniconic gestures, they were likely embodying and sharing their mental model of a list, which gave students something concrete with which to reason.

Teachers are using embodiment to express computation, whether they are conscious of doing that or not. There are things professors are doing in the classroom that are calculated and purely pedagogical content knowledge, and other things that are less conscious and about personal communication styles. For example, the dictionary analogy and accompanying gesture described in Case Study 2 may be a strategy that has been refined over time. It seemed that the analogy was carefully crafted. The ontological metaphor - ”this calls the function” – could be intentional because ”calling” is a standard convention to describe function invocation [58], but could not be intentional for communicating meaning. In other words, the terminology is likely so automatized for the teacher, that they are not intentional for conveying embodied meaning. The gesture of a list described in Case Study 1 may not be intentional but produced spontaneously to help students’ reason. Asking questions about intentionality is essential for criticality and reflectivity of exactly what teachers are communicating by using these embodied representations. Research that has studied the effects of scripted gesture on mathematical learning has generally shown positive learning outcomes. If computing instructors were more intentional in their use of embodied representations, then we may see learning gains.

The analysis presented in this paper is the first step towards understanding how embodiment might affect understanding and opportunities for learning about computing. Our study

points toward a need for a more in-depth investigation of the ways teachers and students use embodied representations and its effects on student learning.

CHAPTER 6
”ON THE REALITY OF TEACHING PROGRAMMING”: INTERPRETING EMBODIMENT IN CS CLASSROOMS

6.1 Introduction

In this chapter, I analyzed computer science (CS) learning by looking at how teachers use embodiment to communicate and structure learning opportunities in order to understand what students need to interpret to learn. Over 30 years ago, ACM Turing awardee Edsger W. Dijkstra argued that “really” teaching computer science is “cruel” [9] and that “computers represent a radical novelty” that simply cannot be understood by metaphor or analogy. To truly understand CS, students must be prevented from attempting to interpret computation in terms of their daily life and physical selves. We now know that Dijkstra’s suggestion is not just that teaching CS is cruel but that it is impossible. All learning is a process of building on existing knowledge [44], and it all begins from our experience of our physical bodies, including that of computer scientists.

However, his speech has likely contributed to Computing Education Research (CER) and the practice of Computing Education (CEd) neglecting the body’s contributions of metaphor and embodiment can make to learning and comprehension of CS. Therefore we have missed out on opportunities to understand what resources support learning, which teaching practices are successful for learning or cause confusion, how do students make sense of computation, etc.

In the previous chapter, I found that professors use embodiment and physicality to explain computation. For example, a professor used a series of gestures to describe the functionality of a list and “act out” iterating through the list. Professors suggest to the class, “Let’s run the code,” and tell students, “Now, we jump to here in the code.” Embodiment is

likely one of the few resources students can use to understand the abstract.

Research on the multimodal practices of teachers found that language and gesture support learning, comprehension, and conceptual understanding [26, 149, 150, 151]. Multimodality refers to the "full range of communicational forms" that students or teachers use to construct and communicate meaning [75]. A speaker's embodiment facilitates the listener's comprehension of speech, particularly when the speech is "highly complex" [28, 101]. In other words, in domains like CS where concepts cannot be directly experienced, teachers' use of embodiment likely creates a representation that students can use to achieve a conceptual understanding.

Just as CS teachers use embodiment to communicate meaning (even if not intentionally), students will interpret and interact with those embodied ideas within activity systems that are socially, culturally, and historically constructed and dependent on their lived experiences[152]. Therefore, students will likely use the embodied communication of their teachers when trying to understand complex programming concepts, such as recursion. However, the use of embodiment may be central to CS classrooms, but it only supports learning if students have the competency or literacy to interpret such embodiment. Consider in our example, would students even understand that the gestures "acted out" are adding elements to a list, and would they be able to attend to perspective-taking?

The challenge with interpretation is that students' comprehension may be challenged by instructional discourse that presents new concepts and uses unfamiliar terms. As students encounter new words, new ways of using language, and new distinctions, they must learn to grapple and reconcile with the "otherness" [153]. When a CS teacher says, "I'm here in the code," they do not mean that they are physically inside the program. Rather, they are using a metaphor that is commonly understood by computer scientists.

I hypothesized how embodiment can create opportunities for learning and, simultaneously, make learning CS difficult because the representations are never surfaced, never explored, and never explained. I study embodiment by analyzing the ways embodied rep-

resentations (i.e., gesture and metaphor) appear and influence CS learning. Drawing from contemporary theories of embodied cognition (e.g., [6, 87, 109]), I used grounded theory to analyze a set of naturalistic video recordings of undergraduate computing professors teaching recursion to their class. I paid particular attention to how the professor used embodied representations - i.e., gesture and metaphor - to support the learning of recursion. We selected recursion because the difficulties with learning the topic are well documented in computing education literature, and it is a topic that is generally agreed upon that every computing student should know [55].

To consider what information and understanding are transported by the use of embodiment to facilitate student learning, I considered the following research questions:

1. How do embodiment and metaphor function as teaching and learning tools about recursion?
2. How do CS teachers describe and gesture about recursion during instruction?
3. How does a teacher use talk and gesture to convey CS knowledge?

These explicit questions about the embodiment of concepts help structure what teachers are striving to teach or communicate and how the embodiment may or may not achieve those learning goals. The first step toward examining the embodiment of teachers is through identifying the kinds of gestures teachers use (because different types of gestures serve different purposes) and the different ways teachers use metaphor. In this chapter, I did not consider sketches since only one professor in my corpus created a sketch (as described in the previous chapter).

This work is about explicating how we teach CS, where we are unnecessarily making it more difficult (by not surfacing our metaphors or by not designing our embodied representations), and the unconscious construction of barriers for those who do not think in just this one way like a computer scientist. Studying the ways teachers use these embodied representations reveals opportunities for teachers to communicate to students in ways that

better support learning [154]. Issues of equity and social justice are both implicit and explicit for understanding how classroom communicative practices impact CS learning. CS classrooms are cultural and social spaces. Therefore, social inequities are easily perpetuated by the use of communicative practices that privilege students with certain forms of knowledge. Participation is political. Given that CS is a science of the artificial, if we can identify these sources, then we can design and think about better ways to design or make "real" CS phenomena.

These insights likewise extend our understanding of how metaphors are useful for CS learning, what resources students use, and how they make sense of computation. Researchers have surveyed professors to understand the kinds of metaphors teachers use while teaching. However, instead of surveying professors, I observed them and found that they use different metaphors. Likely, the metaphors I described in this chapter are more authentic because they are naturalistic. It is possible that surveying teachers derives more intentional, thoughtful metaphors, but I was able to capture the metaphors that are created on the fly.

After presenting the data collection and data analysis, I present the findings and case studies. The findings are meant to be descriptive and interpretable and are not exhaustive lists. I conclude with the argument that interpretation and learning CS are difficult because of the embodiment and metaphors used. We consider the implications of this argument specifically to assert that we need to do a better job of designing our metaphors and embodiment.

6.2 Multimodality and Communicating Information

The discussion of embodiment suggests that embodied representations are multimodal and, therefore, communicate meaning. An important strand of research has investigated the ways representations are used in classrooms to best support and facilitate learning [26, 75]. This research has investigated topics ranging from the "semiotic potential" of representa-

tions, how students exploit representations to support meaning-making, and the pedagogical challenges with dealing with representations. Significantly, this research has led to a multimodal perspective on learning and an assertion that representations actively mediate and shape knowing and reasoning and play a defining rather than a supporting role in understanding.

Multimodal theories explain that different representations are communicating different but important information. To learn with the representations, students need to coordinate and fluently transfer between the representations. This research has argued that making-meaning with representations involves the simultaneous abilities to ”think with” and ”think through” a representation. Meaning-making is defined along three dimensions [26]:

1. Students developing the ability to recognize, use, and construct accounts of domain-specific phenomena
2. Knowledge-building is considered the use of material and symbolic practices for inquiry
3. Reasoning is enhanced by the process of students' constructing and interpreting representations

Ibrahimi and colleagues argue that the two processes of 'thinking with' and 'thinking through' multimodal representations require a deep understanding of the contribution of embodiment within meaning making [75]. They illustrate this point with the example of a teacher explaining day and night. They gestured by spinning their finger around while describing the number of hours required for Earth's rotation. They suggest that the combination of the gesture and speech provides a sense of pace that is not communicated through only speech. The different representations offer different affordances, that when combined and ensembled, allow different forms of meaning to emerge. They conclude that an examination of how the body acts as a representational resource is needed to fully understand the ability it has to enhance teaching and student learning.

The challenge with coordinating and using the representations teachers create is that students need to be able to interpret the representations.

Interpretation requires students to determine the referent. Peirce explains that there are three things involved whenever there is a representation [155]:

1. The referent or whatever is represented
2. The referring expression that represents the referent
3. The interpretation that links the referring expression to the referent

Therefore, if a teacher gestures or uses a metaphor, students need to be able to answer, ”to what are they referring,” to interpret it. Complicating interpretation is the fact that it is an inherently contextualized activity: “students must understand the relation between the representation and the domain that it represents and must understand the representation within the confines and definitions within the domain” [26]. Ainsworth continues that interpreting is a particularly difficult task since this understanding must ”be forged upon incomplete domain knowledge” [26].

6.3 Methods

The goal of this work is to document teachers' embodiment and hypothesize about how easy or difficult it is for students to interpret. I was interested in understanding how teachers communicate information about recursion and what where communication might break down. Specifically, to consider interpretation, I consider what kind of gestures and metaphors were used, how they were used, and what might have guided the teacher in using them. This allows us to understand what knowledge is communicated and through what mechanisms to facilitate students conceptual understanding

Based on research from McCauley, I operationalized learning of recursion along three dimensions: (1) comprehension, or describing recursion and recursive solutions, (2) evaluation, or tracing a recursive solution, and (3) construction, or writing a recursive solution

[50]. Moreover, because I'm focusing on those dimensions, it allows us to make more general claims about how teachers use embodiment to support CS learning.

I present the findings in the style of detailed episodes; this work is meant to be descriptive and interpretive. Furthermore, because of the nature of this work, I do not provide exhaustive lists or categorizations.

6.3.1 Data Collection

Data Sources

Similar to Lewis's 2014 ICER paper, I am not testing a hypothesis, but using a grounded approach to identify how embodiment was used by teachers in classrooms [156]. The methodological approach was influenced by learning sciences research that argues "the contributions of embodied representations to teaching and learning is found within practical work in teaching" [75]. Therefore, the data corpus consists of naturalistic video recordings of teachers as they teach recursion to their students. The recordings were filmed during typical class time.

In total, I gathered 227-minutes of video data from six undergraduate professors. I emailed professors at universities that are allowed to video record their classrooms, emailed CEd researchers in industry or at data companies, scoured MOOCs, and online video databases to create our video corpus. In all, I received or examined 33 videos.

Since this study was dependent upon those who responded and those who were able to send us video recordings, it provides a potentially skewed view of what happens in computing classrooms. This is a limitation for two reasons: First, all the professors were middle-aged white or Asian males teaching at different universities in the United States of America. Second, the professors in the videos are known as excellent computer science teachers. However, this sample is representative of teachers and teaching practices that are rewarded by institutions – these professors were all tenured at prestigious computer science departments – and modeled by others. Each course taught a different imperative language,

and all of them were “conventionally structured” [50], consisting of Socratic-style lectures conducted in an auditorium and practical laboratory work in other sessions. The videos show only the professor, but some students’ voices are intelligible.

Moreover, because I did not create these video recordings, it was not possible to dictate the framing of the video or what was in focus. However, the framing does provide insight into the aspects that other people think should be given importance. I was also unable to collect artifacts from the classes or interview the professors and, therefore, cannot make any claims regarding intentionality.

Videos

While all of the classes were typical college-level length (50-60 minutes), not all the professors spent the entire time teaching recursion. Some professors did not trace any recursive functions or focused only on generation. Table 6.1 is a breakdown of the videos, detailing the demographics of the professor, how long each video was in minutes and how much of that time the professor spent teaching recursion, what other lessons/topics they taught if they taught more than just recursion, and what they taught about recursion, and how long the lesson about recursion was. I personally identified the professors’ race/ethnicity and gender. Therefore, I used certain labels, such as “Asian-passing” and “white-passing,” in the event that they do not identify as Asian or white. I also included the label “male-presenting.”

6.3.2 Data Analysis

Case Selection

After the data was collected, data analysis began by viewing all the videos from start to end. Content logs were created for all video data, where special attention was paid to any interesting details and episodes. These included instances where the teachers used unusual language to describe a concept, where they referred to the code as “it,” and where spatial

Table 6.1: Professors' demographics and video information

Professor	Demographics	Topics Taught in Lesson	Length of Video (in minutes)	Time Spent Teaching Recursion	Parts Taught
P1	white-passing male-presenting	recursion	46 minutes	46 minutes	comprehension, generation, evaluation
P2	white-passing male-presenting	recursion and dictionaries	48 minutes	33 minutes	comprehension, generation
P3	white-passing male-presenting	recursion	46 minutes	46 minutes	comprehension, generation
P4	white-passing male-presenting	algorithms; recursion; search; sort	88 minutes	11 minutes	comprehension, generation
P5	white-passing male-presenting	recursion	52 minutes	37 minutes	generation
P6	Asian male-presenting	object-oriented programming; recursion; inheritance	58 minutes	7 minutes	comprehension, generation, evaluation

language and extensive gesture sequences were used. The language was eventually coded as metaphors. I then created multimodal transcripts for 15% of the episodes.

Next, I chose two contrasting cases for each dimension (comprehension, evaluating, and construction) to begin hypothesizing about the kinds of embodiment that were used and their purpose. These contrasting cases focused on (1) gesture production – an instructor who gestured more than another – (2) and language use – an instructor who used more interesting language than another. It became clear that instructors used embodiment to fulfil similar objectives and purposes and used similar types of embodiment. It also became apparent that how much an instructor gestured seemed to vary with each individual.

Then, for each of these cases, I coded the data, beginning with the transcripts. I began by conducting line-by-line coding. After discussions with the research team, we decided to use embodiment as a framework. We then concluded by coding the rest of the data set, iterating on the codebook as needed. The rest of this section describes the coding process.

The cases presented in this chapter were chosen because they showcased examples of teachers creating extensive sequences of gestures and using heavily metaphorical language.

Coding Gestures

I used the methods specified by Trafton et al. [157, 158] and Stieff and Raje [159] to analyze teachers' gesture production. Using Trafton et al.'s framework, I identified four kinds of gestures: (1) beat, (2) deictic, (3) iconic, and (4) non-iconic.

Beat gestures were "typically brief, motorically simple gestures" [157], including gestures that go along with rhythmic language, communicative gestures (e.g., thumbs up), and personal gestures (e.g., touching one's nose). Gestures were denoted as deictic if "there was a directed, explicit pointing action, usually involving a finger or forearm, and a purposeful direction toward a display or item in the environment" [157], including pointing to a specific location, thing, or line of code, and typically followed by a demonstrative (this or that). Iconic gestures were any gestures that "acted out" a concrete sequence; specifi-

cally, these were gestures “that had a strong relationship to the semantics of the utterance, or ‘acted out’ what was said” [157]. Lastly, non-iconic gestures included gestures that could not be placed into any of the other categories, “were a mix of metaphoric gestures and non-codable gestures (they were not iconics, beats or deictic gestures)” [108]. Some scholars have argued for just collapsing iconic and metaphoric gestures into one category, but we decided to keep them separate. Some of these gestures seemed to “act out” abstract concepts, like process. Knowing a teacher’s intentionality, such as asking them what they thought a gesture meant and understanding their underlying metaphors and mental models of constructs and syntax, could help parse it out.

I used a two-step gesture coding scheme. I first reviewed the video recordings with the sound off and tagged every occurrence of a gesture. Following this, I reviewed the recordings a second time with the sound turned on and the concurrent transcripts. During the second viewing, I classified each gesture using the taxonomy previously described.

Coding Metaphor

I used grounded theory to code the transcripts of three videos [46]. I began analysis by line-by-line coding and looked for any noteworthy patterns in the data. As I was initially interested in spatial language and metaphors, I paid extra attention to spatial words (e.g., here, before, etc.) and spatial metaphors. During this first pass, I also coded concepts that were “interesting,” particularly words or phrases that we felt “seemed spatial” (e.g., calls, returns, etc.) or any language that seemed anthropomorphic or could be used to personify. On the basis of this initial coding, I identified dozens of codes related to an emergent theme of how professors use embodiment while teaching.

After discussions, we noticed that many of the words seemed to be metaphorical and anthropomorphized code. After a literature review, we began to notice that all the codes we tagged were about embodiment, specifically in relation to metaphor use and perspective-taking. After the literature review, we then reiterated our codes by tagging metaphors and

other categories as different kinds of metaphors (personification, whatever the other ones were). In addition, I tagged any time the professor used different pronouns that seemed to refer to the computer; this eventually became the categories actor-perspective and code-perspective.

To determine whether or not the words or utteraneces were metaphors, I used a coding scheme described by Jeppson et al. and the Pragglejaz Group which provides explicit criteria for categorizing a word as metaphorical [82, 160]. This method is a list of steps to determine whether a lexical unit (word or phrase) is a metaphor (see Figure 6.1).

1. Read the entire text–discourse to establish a general understanding of the meaning.
2. Determine the lexical units in the text–discourse
3. (a) For each lexical unit in the text, establish its meaning in context, that is, how it applies to an entity, relation, or attribute in the situation evoked by the text (contextual meaning). Take into account what comes before and after the lexical unit.
   (b) For each lexical unit, determine if it has a more basic contemporary meaning in other contexts than the one in the given context. For our purposes, basic meanings tend to be
   —More concrete; what they evoke is easier to imagine, see, hear, feel, smell, and taste.
   —Related to bodily action.
   —More precise (as opposed to vague)
   —Historically older.
   Basic meanings are not necessarily the most frequent meanings of the lexical unit.
   (c) If the lexical unit has a more basic current–contemporary meaning in other contexts than the given context, decide whether the contextual meaning contrasts with the basic meaning but can be understood in comparison with it.
4. If yes, mark the lexical unit as metaphorical.

Figure 6.1: The approach to determine if a word or utterance is a metaphor [160].

Segmentation

To make claims about what purpose embodiment might serve or the goal behind its use, I segmented the data inspired by a discourse analysis or a move analysis. A discourse analysis methodology can be used to unpack the intent, as a move is considered to be a

distinct shift in focus or a change in topic or purpose. A move has a goal or intentionality and moves the unit of analysis away from an utterance.

This focus segmented teachers' talk into chunks that delimit units of speech produced according to what that speech is doing in the interaction.

Categories of teachers' moves were developed and refined by two researchers using an iterative process that involved an analysis of the nature and intent of teachers' statements. Each move was assigned to an existing code or to a new code as necessary. Where there was disagreement or ambiguity, the researchers discussed this and decided whether the utterance warranted a new category or pointed to a need for clarification or expansion of meaning within an existing category. The coding categories were refined to the point where all utterances would fit within the coding system.

6.4 Reflexivity

Lewis argued that researchers should make their epistemological assumptions clear when reporting on qualitative analyses. We are computer scientists from different cultural backgrounds [156]. Admittedly, the first author, an upper-middle class Black woman, is personally invested in this topic because she almost left CS since she could not understand many of the metaphors and analogies or some of them were offensive and racist. As computer scientists, this research was conducted from an insider perspective. This made identifying metaphors difficult because we have consented to being indoctrinated into the practices, including the ways language is used. However, our insider status did make it so that we could glean meaning from some types of gestures, that someone with an outsider status may not be able to do.

6.5 Findings

Our analysis yielded several findings related to how teachers use embodiment and the kinds of embodiment used. Across all the data analyzed of teachers teaching recursion in class-

rooms, I found 1136 instances of embodiment demonstrated through metaphor and gesture. In the rest of this section, I describe the different types of gestures and metaphors teachers used when teaching recursion. In the next section, I present three vignettes to illustrate how gestures and metaphors were used by teachers.

6.5.1 Gesture Production

I computed the frequency of the type of gestures produced. As expected, beat were the most common type of gesture (61.8% of all gestures coded), followed by noniconic (18.0%), then deictic (11.3%), and last, iconic (8.8%).

Deictic Gesture

As previously mentioned, deictic gestures are pointing gestures. Typically, all teachers used deictic gestures to orient and point to information when explaining or communicating ideas. One teacher used a series of deictic gestures to trace a code execution. It should be noted that only two professors traced a code execution; the other professor that did trace sketched the code trace (described in the previous chapter). As a pedagogical function, deictic gesture seemed to orient student attention to salient features, thereby "refining and qualifying" the information communicated through other modes.

Iconic Gesture

As stated in the previous section, iconic gestures are gestures that "act-out" a concrete referent. These gestures appeared when teachers used a concrete analogy or example when trying to define or explain what recursion is conceptually. For example, one of the teachers compared recursion to employing "simpleton" workers who are only able to accomplish one thing. The teacher then made a series of gestures where they "acted out" a worker counting the number of people in a row. Another example is of the professor who used the analogy that recursion is like looking up words in a dictionary (see Chapter 5). They used a

series of gestures through which they imitated a person searching for words in a dictionary.

Noniconic Gesture

As stated before, noniconic gestures refer to gestures that could be metaphoric, or represent abstractions and computation, or could be nonsense. As previously indicated, I differentiated between iconic and noniconic gestures by assuming that iconic gestures refer to something concrete.

The professors primarily used noniconic gestures when referencing computation. For example, Figure 6.2 illustrates a series of gestures that a professor made that seem to be about “the call stack.” He cups his hands as if to show the chunks of memory in the stack and concludes the gestures by pulling his hands apart as if showing the entire stack. As a pedagogical function, noniconic gesture seemed to provide something “concrete” for students, which then gave them something to reason and hypothesize with and about.

(a) Talk: you know when you call functions in the memory of the computer it allocates*: a little bit of memory
Action: cuffs hands, and puts one on top of the other

(b) Talk: it knows where to jump* to and stuff it's
Action: continues to put one hand on top of the other

(c) Talk: called a call stack* you know when your program crashes and you get the list of
Action: flattens palms and moves hands away from each other

Figure 6.2: Teachers’ gestures while describing memory and the call stack. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image.

These gestures make us question what it means to ‘act out’ recursion or computation. Are they acting out the program (the actions defined by the programmer)? Are they acting out the Python interpretation (e.g., how the stack pops in recursion)? The processor’s action

(e.g., where different variables are in memory)? There are so many layers to an executing program, and the teacher is likely trying to explicate a level that the student is confused about or is getting in the way of success.

Lastly, we found five professors used an oscillating, cyclical gesture, where the professor would use either one hand or both hands in a cycling manner. This gesture was used in conjunction with utterances like ”recursion keeps repeating again, and again, and again” or “...dealing with recursion and its mind blowing cyclicity.” The one professor (P2) that did not use this gesture did not talk about the recursive’s cyclical nature. We describe this gesture in more detail in the next section.

6.5.2 Metaphors

We differentiated analogies from metaphors because we consider the two to serve different purposes. According to Brookes and Etkina [81], the critical difference is that analogies suggest the source domain *is like* the target domain (e.g., life is like a box of chocolates), whereas a metaphor suggests the source domain *is* the target domain (e.g., they are a shining star). In a study about the use of metaphors by physicists, Brookes and Etkina states that when physicists need to assert something, they need *is* more than *is like* in their reasoning process [81]. The author concludes that “is” represents a fundamental trait of how knowledge is generated.

Metaphors, such as *calls*, *prints*, and *runs*, do not seem to be metaphors. Professors used these frequently throughout their classes like they were just common knowledge and everyday words. They are likely dead metaphors or metaphors that are so common and widely accepted that they have lost their original metaphorical connotation.

I found that professors used metaphors and physicality to represent computation or computational processes. Table 6.2 displays a list of some of the metaphors identified. Every professor used these metaphors. However, only one professor (P2) used the metaphors *bounds*, *reduces*, and *unwinds*.

These are metaphors because none of these actions are actually happening. A variable is not *bound* to anything, and a function does not actually *run* or *print*. Metaphors anthropomorphize computation, which suggests that computation has behaviors, intentionalities, and goals. For example, a variable keeps changing *until it gets to one*. This metaphor suggests that code has agency and directionality, that is, the code can go to some destination.

I also found that a teacher use spatial metaphors while describing a code trace (e.g., *so it goes to here*). The spatial metaphors describe where the recursive process is both virtually and physically - or where in the lines of code.

In this regard, metaphors serve three pedagogical functions:

1. Students can rely on their physical and lived-experiences to make sense of computation.

2. Metaphors provide a language for which to talk about abstractions.

3. They provide a way for students to conceptualize and reason about abstractions and their behaviors.

As previously stated, Professor P2 used the metaphors different metaphors when referring to variable assignment and recursive process, *bound* and *unwinds*. The use of these metaphors could be cultural; although the professor taught at a school in the U.S., the professor is Canadian.

The professor said, “This variable is bound to 3,” whereas the other professors expressed the same idea by saying that a variable is “equal to” or “gets” a certain value. Likewise, the professor referred to the recursive process that happens before the base case is executed as *unwinding* while other professors referred to it as cyclical and repetitive.

It should be analyzed how the use of different metaphors might lead to different conceptualizations. Maybe using a metaphor, such as *bound*, could be a better way to describe variable assignment, as saying “equals to” tends to lead to misconceptions about the concept. A “binding” suggests that a thing is confined to something, which may be more

effective at helping students realize that reassigning the value of a variable does not add to its initial value.

However, this example highlights two important findings associated with metaphors. First, metaphors are cultural. A student can only use a metaphor, like *bounds*, if they have a similar referent as that of the professor. Second, many of the metaphors have a limited scope; they only describe one aspect of recursion or computation. *Bounds* might highlight variable reassignment, but it does not necessarily suggest that variables can only have a single value. These metaphors are not necessarily completely wrong, but teachers need to be aware of their limitations and how they might be confusing.

Metaphorical Construals

The analysis led to the identification of two unique relationships formed between the professor and code. I found the professor switched (sometimes fluently) between an actor-perspective (e.g., "then we go here" or "then I go here") and a code-perspective (e.g., "then it goes here"). Pedagogically, this allows the teacher to "blend" "subject with object" to engage more deeply with code or logic, as way to reason and hypothesize about process and logic.

The actor-perspective included the pronouns you, me, we, and I. It was not obvious who the professor referred to when using those pronouns, therefore we included them in the same category.

The code-perspective included the pronoun it. Again, it was not obvious who the professor referred to when using this perspective. In some moments, the professor seemed to refer to process, in another the function.

I describe these issues further in later case studies.

The use of different perspectives creates this metaphorical construal of the computer or process as having agency. Professors used these metaphorical construals when they tried to help students understand the code's logic. As a teaching tool, students can 'role-play' as if

Table 6.2: List of Some Metaphors

Metaphor	Reference	Example
bound	variable assignment	"So n is bound to 4." "that value, right there, of 3. So 3 is now bound to n."
call	function invocation	"We know that functions can call other functions, but functions can call themselves, oftentimes, to create some sort of useful effect."
reduces to	variable changing	"Well that call to fact of 1. So that reduces to return 2 times 1. And who called for that?"
unwinding; unwraps	recursive process	"so long as I have what I call a base case, a way of stopping that unwinding of the problems, when I get to something I can solve directly."
run	execute a set of instructions	"and then run V and let me show you the Vblock it says turn left 45 move 25 steps up and then choose a random item from that shapes list and run it so this part is kind of a preview of things you'll see later because"
return	stop execution of current subroutine	"from position p okay okay so this is going to return a list of all the arrows all the moves you've got so"
prints	display output	"It's just going to print something. But it is going to take input, like a number n"

they were doing the action and reason about logic using their own lived-experiences.

Professors also used different perspectives while doing a code trace as a way to navigate code. This was sometimes combined with spatial metaphors, which seemed more like a spatial strategy. That is a way to structure space and embody a code trace.

6.6 Case Studies

In this section, I present three case studies in the style of vignettes to illustrate how gesture and metaphor were used to explain computing concepts. The three case study vignettes are from two different professors (P4 and P6). The aim is to understand what guides the teacher's use of gestures and metaphor and what information is being communicated.

Case Study 1: Metaphors to Explain Recursion. In this case, the intent is to show how a professor uses metaphors through words and gestures to help students understand recursion conceptually. Specifically, we highlight the many learning opportunities the professor creates, but how they also may, unintentionally, be conveying recursion as a loop through their use of embodiment. The goal is for students to form viable mental models of recursion. However, the metaphor and gestures used by the professor, arguably, convey recursion as cyclical. From our observations of CS classrooms, the professor uses typical pedagogy, introducing concrete examples, comparing recursion to iteration, and defines recursion typically.

The class is a data structures and algorithms course taught using Python programming language. The teacher stood at a lectern in the front of the class with a laptop connected to a projector. On the opposite side of the podium is a large television screen. From this 11-minute class we describe a 4-minute video clip. In this clip the professor is introducing recursion to the students. The professor asks the class to "go back to week 1," when they discussed a phone book application, "with a programming construct that, at the time, we highlighted and called a loop."

Instead of immediately discussing recursion, the professor introduces a concrete exam-

ple. He walks away from the podium towards the TV screen with pseudo code of the phone book algorithm. He points to the code and says,

> ”we highlighted and called a loop, go back to line 3. So that you can do something again and again. This is an example of what’s called iteration,where to iterate just means to loop again and again.”

This utterance co-occurred with a noniconic gesture (see Figure 6.3), with the professor moving his hand in an oscillating, circular motion, where the hand rotates or traces a circular movement when he says ”iterate.” A common pedagogical technique professors use is to compare recursion to iteration, and transform iteration to recursion.

Talk:where to iterate* just means to loop again and again.
Action: moves right hand in cyclical gesture

Figure 6.3: Professor making a cyclical gesture.

This circular movement is a gesture that we observed in all of the videos. Professors use this circular movement throughout their lesson when referring to function invocation, the cyclical nature of recursion, or, in this case, iteration. As previously stated, noniconic gesture are likely good learning aids because they can replicate movement, or “simulate action.” Specifically, when students attend to these types of gestures, a shared space is created.

Therefore, this gesture may be a representation of the ways that the professor envisions invocation, recursion, loops, functionally, work, and by using them, he is sharing his mental model of those programming constructs with the class. However, these gestures only have a meaning if they are ”meaningfully connected to the physical action being learned are less likely to help”.

But motivating these gestures is likely a certain mental model, an underlying metaphor. As a gesture, it could easily be seen as a random hand movement, just going along with the rhythm of his talk. But if we think about how ”loops”, ”function invocation,” or ”recursion” are modeled as typically something cyclical like a loop, then that sort of gesture could be the teacher’s mental models. But, for a novice student learning recursion for the first time, this gesture may be difficult to extract a meaning from. Understanding embodiment may require a way of ”seeing” that is accepted in a community of practice.

Moreover, the use of this gesture to represent three different constructs, could easily conflate the meaning. If a student were to understand them, seeing the gesture used to explain different programming constructs can make it as though all the concepts are the same or closely related.

The professor continues with the lesson saying,

> ”there’s an opportunity to design this algorithm not only differently, but perhaps better, right?... and get rid of this iteration and see if I can’t solve the problem more elegantly, if you will, a better design.”

The professor uses an ontological metaphor, ”elegant”, making a value judgment about what constitutes ”good code.” Using this metaphor also personifies recursive solutions, as if a block code could be ”pleasing and graceful in appearance or style”, and making comparisons between recursion and iteration, that for *real* computer scientist, recursion is conventionally and culturally a better form of iteration.

The professor gives an example,

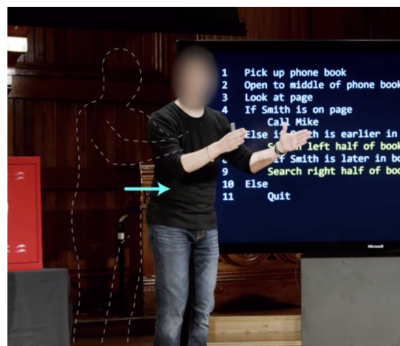
”I’m telling you, if you want to search for Mike Smith in a phone book of this size, mm-mm. Search for Mike Smith in a phone book of this size. And then the next step of that algorithm becomes search for him, in a phone book of this size, this size, when you keep halving the problem.”

Figure 6.4 demonstrates how this speech is more communicative when it is augmented with gestures. In this example, the professor uses physical space in his iconic gestures to communicate different meanings. He first places his hands far apart to show length when saying

”if you want to search for Mike Smith in a phone book of this size”.

Talk: I'm telling you, if you want to search* for Mike Smith in a phone book of this size, mm-mm.
Action: spreads hands apart

Talk: Search for Mike Smith in a phone book of this size*.
Action: brings hands closer together



Talk: And* then the next step of that algorithm
Action: takes a step to the left

Talk: becomes search for him, in a phone book of this size*, this size, when you keep halving the problem.
Action: brings hands closer together

Figure 6.4: Professor uses iconic gesture while describing searching through a phone book.

Then he moves his hands closer to show a shorter length: ”mm-mm. Search for Mike Smith in a phone book of this size.” He then takes a step to the left, but keeps his hands in the same position: ”And then the next step of that algorithm becomes search for him.”

He continues to shrink the space between his hands, pulling them closer together.

Students may not realize that the professor just gave an effective example of a central tenet for recursion, which is to decompose a problem into simpler, more easily to solve problems. Potentially, such “acting out” may be a good learning aid, since novice students typically write recursive algorithms that do not have terminating conditions, or a base case [161]. The gestures added richness to his example.

The professor concludes this example, by telling students that this

> ”is an example of a technique in programming called recursion, whereby you implement a program or an algorithm or code that, in a sense, calls itself.”

The statement, a program or an algorithm or code that in a sense calls itself, is layered with embodiment. In our data corpus, every teacher defined recursion as ”a program/function that calls itself.” It’s a common phrase; even anecdotally, textbooks use the same phrase and so do we. But, what exactly does it mean to ”call itself”? The statement personifies recursion by referring to ”itself” and suggesting that a function or code can make a *call*. How can a function have agency or intentions?

Moreover, *call* is metaphor used to describe function invocation, but what exactly does it mean to ”call something” and where did that term even come from? Code, then, can’t just be lines on a paper if teachers are requiring you to be in it or calling it. We *call* things to communicate with others that are at a distance from us. In contrast, we speak or whisper to listeners who are near us. Why would a function that is referencing *itself* need to *call* it? Why would it be distant from itself? What work does the phrase “in a sense” do in this utterance? Perhaps the professor is aware that he’s drawing on a metaphor, and uses that phrase to mean “I am not speaking in a literal sense here.”

The student must be able to interpret these metaphors and gestures being used by the

teacher in order to comprehend the instruction. Educators are in the position of beginning to understand the kinds of foci that can facilitate understanding CS classroom practice. To help educators we need to develop a catalogue of the common metaphors that happen in the CS classroom and then to uncover how students make, or do not make, sense of these communication methods.

Case Study 2: Writing a Recursive Function. In this case, I intend to show how a professor uses embodiment, specifically perspective-taking, while writing a recursive function. Professors switch between perspectives as if they're role-playing different roles. They likely use perspective-taking to help students develop an embodied understanding of the logic of the code to write a recursive function. In other words, they are trying to help students form analogies based on their lived-experiences to reason about the code's logic. This is a good approach since, as Pirolli argues, students need a template or analogy to write recursive programs [54]. However, teachers do not coordinate between the roles which can hinder students from forming an embodied understanding.

This 12-minute video clip is the same professor and class session from the first case. The professor asks the class how they might write a program that prints a pyramid, like the one from the 1985 Nintendo game, Super Mario Brothers. Recall that in this classroom, there is a lectern connected to a projector and a large monitor.

The professor writes the algorithm to produce a pyramid using iteration. He then walks towards the monitor that is displaying an image of the pyramid from Super Mario Brothers. He begins to "decompose the problem" by describing the pattern he notices about the pyramid:

> "there's this common structure, right? And if we look at the pyramid in isolation, what is the definition of a pyramid of height 4? Well, arguably, it's a pyramid of height 3 plus 1 additional row. What's the definition of a pyramid of height 3? Well, it's a pyramid of height 2 plus 1 additional row... That's a recursive definition of just a physical object or a virtual object."

The professor is using a concrete example to explain the recursive case.

He then identifies the base case,

> ”Now, at some point, I need a special case, at least one height. What is a pyramid of height 0? Nothing, right? So long as you have a so-called base case, where you manually say, oh, in that specific case, just don’t do anything, and you don’t recursively call yourself again and again, we can use this principle of code calling itself.”

Again, the professor makes the same cyclical gesture when he says ”recursively,” possibly sharing his mental model of recursion. The professor also uses perspective-taking, acting as though he is the recursive solution when talking about the logic of code, and giving the agency to the class when talking abstractly about the code, concept-based. He is trying to create an embodied understanding of recursion by attempting to relate it to lived-experiences. He’s using his own experiences and judgments (i.e., how would he do something) to justify the logic of the recursive function. He’s trying to identify with the program.

He walks back to the lectern and begins writing a recursive solution to draw a pyramid: ”My goal now is not to just use nested loops, but to define a bigger pyramid in terms of a small pyramid.” He simultaneously uses his hands to gesture something shrinking when saying ”bigger” and ”smaller”, a way to show a problem decomposing. He asks the class, ”How do I draw a pyramid of size 4 in English?” A student responds, ”Draw a pyramid of the size 4 minus 1.”

The professor responds:

> Yeah, draw a pyramid of size 4 minus 1, or a pyramid of size 3. So how do I express this in code? Well, wonderfully in code, this is super simple, h minus 1. That will draw me a pyramid of height h minus 1, or 3 in this specific case. Now, it’s not done the program, right? I can’t possibly just compile this and expect it to work because this seems like it’s just going to call itself endlessly.

Notice, the professor switches between perspectives, saying “that will draw me a pyramid” to “now, it’s not done the program.” Every professor switched between perspectives fluently and with no discernable systematicity.

It’s not clear what roles the professor is role-playing or asking the students to role play. In this quote, the professor seems to role play as someone who owns the code, when earlier he was the code and then the students were the code. Meanwhile the “it” seems to represent the recursive process. The professor writes the code: draw(h-1). Then he says the code is ”obviously not right, because this seems like it’s just going to call itself endlessly”, again using the same cyclical gesture and code-perspective.

The professor then has the class consider the base case.

> Well, what’s a pyramid of size 3, 2, 1, 0, negative 1, negative 2, right? It would go on endlessly if I just blindly subtract 1. So I need that base case. Under what circumstances should I actually not draw anything? (inaudible student response) Yeah. So maybe if h equals equals 0, you know what? Just return. Don’t do anything, right? I need a base case, a hard-coded condition that says stop doing this, this mind-bending cyclicity again and again.

Again, the professor is trying to help students reason about the logic when a recursive invocation should terminate, using their own lived experiences. He switches between perspectives with no discernable understanding why or what role one should play. In the same instance (“it would go on endlessly if I just blindly subtract 1), “it” is the recursive process and now the professor is the code.

Teachers using embodied experiences and perspective-taking is typically considered a valuable approach because it helps students make sense of new information by relating it to their pre-existing and lived experiences. For example, Ibrahim-Didi and colleagues describe cases of teachers very purposefully coordinating different perspectives and roles for students to play to form an embodied understanding of how day turns to night [75]. However, in our corpus no CS teacher was purposeful with perspective-taking. For CS, this

is especially challenging since students don't naturally think recursively. No student will naturally think a pyramid's height is one plus an additional row. Therefore, students likely need systematicity in perspective-taking to help them form an analogous example to write recursive functions.

Every professor in the data corpus used perspective-taking to explain code logic. This suggests that this is likely a practice passed from one professor to another. Nobody has considered how professor's use perspective taking while teaching, likely because CER has not used embodiment as a theoretical lens.

Case Study 3: Tracing a Recursive Invocation. In this case study, we identify how a professor used embodiment to trace a recursive invocation. Specifically, we address (1) the professor's use of perspective-taking and deictic gestures to orient the class and help them navigate code and (2) the professor's use of perspective-taking to role play as different elements in the notional machine. Tracing, or simulating the program execution, is a fundamental skill in computer programming that supports the comprehension of a task. Sorva argues that novices need concrete tracing, where one tracks specific values and how those values change [64].

The professor in this case study is a different professor from the previous two case studies. This professor was one of two in our data corpus that provided a detailed trace of a recursive invocation. Both professors did so out of response to students' questions.

During this 4-minute video clip, the professor is introducing recursion to the students. The class is a data structures and algorithms course and taught in Python. The teacher stood at a podium with a laptop connected to a projector.

The professor asks the students to recall an in-class exercise from the first week, telling them that it was actually about recursion. Similar to the professor in the first case study, this professor gives the following definition of recursion:

> ”But recursion is this idea of... We can have functions, functions that are running code. But, functions can also call themselves. We know that functions

can call other functions, but functions can call themselves, oftentimes, to create some sort of useful effect."

The professor goes straight into an example to calculate the factorial of some number. McCauley argues that many teachers use factorial as a simple example of a recursive algorithm [50]. Factorial is an abstract conceptual model that Wu argues helps students form viable mental models of recursion [52].

On his computer, the professor types the code to calculate the factorial of some arbitrary value using recursion, a function with three lines of code. While writing the code, he uses the same oscillating gesture while uttering, "And so the recursive idea that I can use here is the factorial of some value n."

The professor asks if any students have questions, and one student asks, "The last three lines, you don't really need them, right?" Figure 6.5 is an image of the professors code, with the three lines the student refers to in a white box.

Figure 6.5: The professors code, with the three lines the student refers to in a white box.

The student clearly has an "odd" mental model of recursion, likely misinterpreting what "a function that calls itself" means. Likely realizing that the student does not understand recursive execution, the professor begins a code trace.

> So you do need these three lines, and the reason is this is handling the logic of what the function is doing. So the first thing my function is doing is checking to see if I've hit my base case. If I'm calculating the factorial of 1, I'm just going to pre-program into my function the answer to that is 1. But if I'm trying to calculate the factorial of anything else–presumably something larger than 1– but you should probably add checks to handle things less than that, then to calculate the factorial of 5, for example, I'm going to take the number 5 and multiply it by whatever I would get by taking the factorial of 4 of n minus 1."

In the professor's explanation, he primarily uses an actor-perspective, referring to himself as doing the actions. However, his role changes. At some points, the professor is the owner of the code, at other points the professor is the code, and at another point the professor is the process, he seems to be playing the role of variable (I'm going to take the number 5 and multiply it). Even "you" is the code at some point.

It could be that the professor is playing different roles in program state. To trace, a programmer must keep track of the program state. Perkins et al. describe the program state includes the elements of the notional machine: "variables, objects, references, function activations, and so on" [162]. As a teaching tool, switching between these roles seems promising, because it could help students understand the ways these different elements change during execution.

After that explanation, the professor asks again if any students have any more questions. One student asked, "How does the computer know what the factorial n minus 1 is?"

Likely realizing that the students still do not understand how recursive functions executes, the professor continues the code trace. The professor states, "So now we're running factorial one more time," repeating the oscillating gesture. Notice the professor is using an actor-based perspective; acting like him and the class are the computer. He also uses metaphor, acting as if the function, factorial, *can run.* Running is an extension of the body. Something that people and animals do. How is it, then, that an abstract concept can *run.*

The professor continues: ”How does the computer know what the factorial of n minus 1 is? It calls the function again.” Here, the professor uses *call* a metaphor we discussed in the first case study. However, notice the fluent - and random - transition from actor-based to code-based; a way to ”step-out” and give the computer agency.

Then, the professor switches perspective again, ”So now we’re running factorial one more time. We’re calculating factorial of 4. That’s not one 1. We’re going to return 4 times the factorial of 3.” Co-occurring with this perspective-taking was a series of deictic gestures. The professor used his mouse to highlight each line of code he was referring to. Pointing directed the ”visual attention” of class to the specific lines of code.

Notice the professor uses the metaphors ”we’re going to return 4 times the factorial of 3.” First, is the metaphor *return*, which is meant to imply halting a subroutine and giving some values back. However, return could be based on a number of metaphors. Second, notice the metaphor ”we’re going to. . . ” which seems to have a double meaning. First, it seems to be construed as a spatial metaphor suggesting both physical and virtual execution (that is, the actual line of code and where the code is in the process; this is where they are headed and this is where they are).

The professor continues: “So now, we’re calculating the factorial of 3, which is 3 times the factorial of 2, which is 2 times the factorial of 1 and the factorial of 1, as these if conditions on lines 9 and 6 will indicate, is just going to be 1.”

In an emergent pattern, the professor uses a deictic gesture and an actor-based perspective when describing the functionality at a code-level. The deictic gesture ”grounds” the code - making it real - suggesting the student is ”here” in the code. However, code is only supposed to be instructions for the computer (computer-human interaction). By ”grounding” the code, it provides markers that students can use to navigate code and understand where ”they are” in an invocation. However, if students can take the perspective of code, then it can’t just be something on a page.

While the professor did not switch between an actor and code perspectives as often as

the professor in Case Study 2, we saw that the professor still randomly switched roles. Interestingly, each of the roles the professor played as were essential elements in the program state that the student needs to keep track of to trace code. However, students likely need to coordinate between these different roles. Tytler and colleagues found that when teachers were purposeful with role play, each role gave students a different perspective and understanding [149]. If students need to keep track of different elements to trace, they likely need to coordinate them. Roleplaying as the different roles, when done purposefully, could be a great way to help students coordinate those different roles.

6.7 Discussion

In this chapter, I analyzed CS learning by looking at how teachers use embodiment to communicate and structure learning opportunities. The goal of the study was to hypothesize about what students need to interpret to learn. Taken together, the findings and case studies expound our understanding of how teachers use embodiment to structure learning opportunities in programming classrooms.

Although I hypothesized that students may not understand teachers' metaphors and gestures, I do not know this for certain. To confirm this hypothesis, more studies are needed on what students attend to, teachers' intentionality, and CER needs more taxonomies. For example, a student learning recursion might be able to easily interpret a metaphor, such as *calls*, while the same metaphor might be hard for a student who is new to learning programming to interpret. Moreover, maybe students don't realize that teachers are switching perspectives. Regardless, there's no harm in teachers being purposeful with how they use their embodiment. It will likely only increase explanatory power. Furthermore, there is still value in being reflective and critical about what teachers are doing in classrooms.

To conclude this chapter, I will reflect on the pedagogical and theoretical significance of the findings. Specifically, I will argue that embodiment and metaphor are critical parts of CS learning, and by ignoring them, we miss the opportunity to design our metaphors

and embodiment for ease of understanding and explanatory power. We need to correct this mistake and embrace the design and use of metaphors in CS learning and teaching.

6.7.1 Pedagogical Function

In CS, we teach things that have no visible embodiment. Nobody looks at memory values to understand arrays, objects, or linked lists. Within this context, CS teachers have the complex task of determining how to embody things that have no perceptual embodiment. Furthermore, they need to select and sequence a range of representations to scaffold students' learning about "the virtual world" [115].

We opened this paper with a reference to Dijkstra's infamous speech in which he considered it childish, "shallow," and "paralyzing" to compare computational processes to physical things instead of understanding them for what they are (i.e., "the radical novelty"). Instead, Dijkstra argued for a replacement metaphor in which we ignore our past understandings because they are "wholly incomplete" [9].

Dijkstra's position was partially correct; computers do represent radical novelty. However, as Smith and colleagues argued, the replacement metaphor sought by Dijkstra—which calls for the "simple addition" of new expert knowledge and the deletion of faulty misconceptions—oversimplifies the changes involved in learning a complex subject matter [163]. Various theories of learning tell us that learning requires engagement and reconciliation with our past understandings and that the insights gained from this process become the foundation for how we gain and construct new knowledge. With appropriate instruction, students' past understandings can serve as anchors for knowledge building.

Teachers' embodiment is naturally occurring, unintentional, and communicates meaning. Teachers cannot help but use embodiment because they are themselves embodied. Students are likely attending to and processing these sequences of complex multimodal representations [26, 27, 75, 152]. Using physical experiences is normal and expected, according to Watt, because it helps people understand a program or computation by allowing

them to “identify with certain programs and see things from [the program’s] point of view” [126]. For example, we found that teachers used embodiment and physicality to communicate meaning about computation. They talked about computation as if programs, functions, and processes can do things, and they stated that the programs behave as if they have goals and agency and can even make *calls* and *run*.

CS teachers’ embodiment could serve a pedagogical function. For example, in Case Study 1, the professor used perspective-taking and a series of gestures as if they were role-playing looking through a phone book as a way to help students reason about how recursion works. The gesture added ”richness” or additional information by acting out the logic. Moreover, professors fluently switched between actor and code perspectives. According to research on data science education, different self-to-object relationships support meaning-making in different ways, including shaping the kinds of questions, interpretations, and conclusions one makes [164, 165]. Role playing that uses perspective taking may equip students to ask questions about the recursive process, such as what happens after the recursive execution stops and how the parameters of the recursive invocation change.

Each representation offers unique ways of understanding abstractions, and the use of multiple representations likely provides students with ways of accessing concepts. Multimodal theories tell us that interpreting embodiment takes the acts of coordinating and moving between many different representations to create understanding. Students must develop the ability to interpret representations; however, they may be unable to do so because of a confusing embodiment.

6.7.2 Challenges with Interpretation

I argue that the metaphors contribute to making teachers embodiment difficult to interpret. Metaphors operate at so many levels. I found different kinds of metaphors, including (1) metaphorical construals, (2) spatial metaphors used to describe both virtual process and physical location, (3) metaphors used to describe function invocation, process, variable as-

signment, etc, and (4) metaphors that underlie gestures about computation. Metaphors give teachers a language through which to talk about computation; in other words, metaphors give physicality to abstractions. Metaphors seem to be neutral descriptions; however, recent research on spoken language and translanguaging in CER has indicated that this may not be the case [121, 122]. Metaphors may add to the difficulty of learning to program and understanding concepts, such as recursion. In the rest of this section, I consider how metaphors might lead to challenges with learning.

I found that teachers use complex, layered, and disassociated embodiment. The statement, "a function that calls itself," is a typical way to define recursion. However, that seemingly simple statement is layered with embodiment. Teachers are anthropomorphizing a function by suggesting that it can *call itself*. Students need to be able to reconcile with that.

I also found that metaphors are likely historically and socioculturally constructed; that is, they have been appropriated and given precise technical meanings that are often closely related but not identical to their everyday meanings. For example, CS is inundated with metaphors that are constantly used in non-CS talk, such as *runs*, *prints*, and *calls*. Teachers use these metaphors fluently without any definition. However, it is likely that precise verbal definitions cannot be produced for these metaphors and a domain-specific literacy or a way of "seeing" is required for students to be able to interpret them.

Moreover, professors use so many metaphorical construals and constructions that, as evidenced from STEM learning literature, serve a significant purpose in teaching and understanding abstract domains, but CS teachers use them haphazardly: they switch between perspectives and roles with no systematicity and without explicitly stating they are. Thus, whether or not the embodiment and metaphor used in these cases can really help students learn abstract concepts in CS needs to be investigated.

Even the gestures about computation may be difficult to interpret. Unlike institutionalized gestures, like pointing to a person or a thumbs up, noniconic gestures referring to

computation are likely not as familiar. These gestures are likely communicating meaning about computation and visualizing abstractions, but they also could be communicating misconceptions or nonsense. These gestures likely have an underlying metaphor as a referent that students need to determine to use for understanding. Manches and syntonicity dude theorized that metaphors underpin the way that individuals conceptualize computing constructs and structure their thinking and reasoning. Our findings support this claim. For example, in the stack example presented in the findings, the teacher might literally be thinking about a stack (as in a stack of plates), which could be the underlying metaphor that is the referent.

The differences between students' and teachers' metaphorical understandings of computation could make it difficult for teachers to communicate. As indicated above, professors use metaphorical construals and constructions that help in teaching and understanding abstract domains but use these without any intentional design. Students are embodied with their own pre-existing knowledge. Teachers, on the other hand, operate within sociocultural contexts where they have already learned and know these concepts and have acclimated and consented to them. Teachers need to be more thoughtful with using metaphors and explain as carefully as possible what they mean by a word and how it differs from its meaning in non-CS discourse.

6.7.3 Designing Metaphors

The different kinds of metaphors mentioned in the previous section suggest design on different levels. The first two kinds of metaphors (metaphorical construals and spatial metaphors) suggest teachers should be intentional with how they design lesson plans and the roles they want themselves and students to play. In the rest of this section, I consider how to design for the latter two kinds of metaphors (metaphors about computation and that underlie gesture).

Sfard argues that "the choice of a metaphor is a highly consequential decision. Different

metaphors may lead to different ways of thinking and to different activities" [166]. Paechter said, "We may say, therefore, that we live by the metaphors we use" [167].

CER should further examine the metaphors or metaphorical constructions that we use to teach CS. We can then consider what conceptions underpin our thinking about computation. By bringing these to the surface, we can determine what works and what does not (i.e., what causes misconceptions, misunderstandings, and difficulties; what do students understand and why).

Bettin's research that describes a framework for designing analogies to describe programming concepts reminds us that designing metaphors for constructs such as recursion is not straightforward [63]. There are multiple processes in which teachers try to motivate understanding/belief in the learner when it comes to recursion:

1. The same function is called repeatedly.
2. Previous function invocations cannot complete until the invocations already made have been resolved.
3. The base case provides a result that allows resolution to begin.

Likely, each of these processes requires their own understanding. There may be some more processes we are trying to indicate in modeling recursion.

We need to make sure the relations within that metaphor can satisfy each of these procedures as their own isolated case. If we can ensure a well-formed metaphorical structure for each process that uses a related source domain, that source domain can be more broadly considered as a holistic representative metaphor.

CHAPTER 7

HOW STUDENTS USE CONCEPTUAL BLENDS, METAPHORS, AND EMBODIMENT TO MAKE SENSE OF COMPUTATION

7.1 Introduction

In this chapter, I analyzed 10 problem-solving sessions in which students in pairs collaboratively solved problems on recursion. The goal was to document the ways space and embodied representations (i.e., gesture, sketching, and metaphor) appear while students make sense of computation and express computational ideas. Sense-making is defined as ”the process of structuring the unknown by placing stimuli into some kind of framework that enables us to comprehend, understand, explain, attribute, extrapolate, and predict” [168, 169].

I investigated two research questions:

1. How does embodiment appear or emerge from the pairs' problem-solving process?

2. How do students make use of embodiment to make sense of and reason about computation, in the context of problem-solving?

I critically reflect on how students learn to program, how students problem solve, and what might support their learning. I use the construct of a *conceptual blend* to explain how students do these mappings [170]. Conceptual blending is a general model for the integration of concepts and the creative construction of meaning. It's an in-the-moment metaphor, perhaps the connection of an invisible computational concept to an aspect of the physical world. It may not be a great long-term metaphor, but it advances sense-making by enabling the student programming to draw upon their knowledge of metaphors and the physical world to gain insight on a computational problem.

This finding is in contrast to prior CEd research that argued metaphors and analogies only served pedagogical purposes and were not ”valuable” for CS learning [10]. However, that research does not contend with learning to program as a messy process. I found that students are trying to navigate these spaces and use whatever tools or resources - like metaphors - are available to them to make sense.

The next section is an overview of the conceptual blending theoretical framework and how it is used in the study. Then, I describe the methods used to collect and analyze the data and present the results of the study. I describe each problem and give a general overview of how the student pair solved the problem, including the types of metaphors used and sketches they invented. Afterwards, I present episodes from detailed analyses.

I conclude by discussing the implications for teaching and CEd research. Specifically, I make the case that while it is true that designing a forever-representation may help students understand a concept, they may be more helped by finding a cognitive toe-hold on understanding the problem in the moment.

7.2 Conceptual Blending

Because of the larger role that metaphor played in the students’ explanations, I chose conceptual blending, a slightly more elaborated theory of metaphor, that would help us analyze the multiple metaphors and the embodiment students used. Conceptual blending is a general model for the integration of concepts and the creative construction of meaning. Conceptual blending is useful for us to describe students’ cognitive processing because it provides a language and diagramming style to describe students’ emergent meaning in the problem-solving process. Students blended resources like the code with their past experiences that result in a metaphor or analogy that they used to reason or ”think with.” Students also complexly layered and blended multiple metaphors to make predictions or explanations. These metaphors reference different constructs like process, execution, variables changing, or the stack.

Fauconnier and Lakoff contended that conceptual blending and metaphor theories can and should be seen as extensions of one another or complimentary [171]. Lakoff and Johnson's metaphor theory proposes that metaphors arise when one domain, or target, is cognitively structured in terms another domain, or source [67]. The mapping allows aspects of the source domain to be transferred to the target domain.

Fauconnier and Turner presented conceptual blending as a refinement of Lakoff and Johnson's two-domain model. One key difference is that the conceptual metaphor theory mainly deals with "entrenched metaphors," where structuring one domain using another is stable and long term (e.g., war is the domain to understand love in the metaphor love is a battlefield). Conceptual blending focuses on local mappings between constructs. These mappings might not extend to entire domains and might be relatively short-lived. Since we cannot assume the metaphors students create in my corpus are entrenched and as they are not likely to be a part of the everyday lexicon, the conceptual blending framework allows greater flexibility in modeling what students are doing.

They proposed that people think in "mental spaces," which are used instead of source and target domains. Mental spaces, or source domains, are concepts that we use in our conversations at a specific moment. We pull in, or integrate, different mental spaces that provide us information to think about and talk about the subject. These mental spaces will often overlap, creating combinations of these spaces, thus creating blended spaces.

Conceptual blending has been used to explain how students layer multiple levels of analogies to learn abstract ideas [172] and model different ways that students reason about the propagation of wave pulses [173].

Enyedy and colleagues used conceptual blending to examine how a student maps her own experience onto a ball to simulate the physics of force and friction [174]. More recently, in CEd, Silvis et al. used conceptual blending to analyze kindergarten students learning to program robots and examined "how they navigated programming's representational infrastructure" [175]. They found that children drew on embodied experiences of

how objects move to program robot routes. They conclude that construing code should be viewed as a conceptual blending.

When we talk about conceptual blending, we are referring to the blending of multiple, disparate spaces or concepts into a brand-new concept or idea. A blended mental space arises from mappings between entities in the two spaces, and the blended space can (and often does) include entities and relationships from one or both spaces. The blended space, which is a hybrid of the mental spaces, has new, emergent meaning not found in any single mental space.

An example of a conceptual blend that we will discuss later is when students making sense of a recursive function realized that it computed an exponent. That observation did not permanently change their definition of exponent. Not all recursive functions compute an exponent. But in that moment, when the students realized that this specific function computed the integer two raised to some power, they made forward progress in their sense-making. They understood something useful in that moment for that function. Maybe the students could forever after think about exponentiation as a recursive process – it doesn't really matter. It's a useful conceptual blend that connects a known process (mathematical exponentiation) to a computational domain (this particular recursive function) to support sense-making.

For my purposes, conceptual blending is a useful theoretical framework for a few reasons. First, it provides a useful language and diagramming style to present our findings: specifically, an explanatory mechanism for how people can bring together (or blend) seemingly disparate spaces to describe how students make sense and gain new insights. Figure Figure 7.1 is a typical diagram, where there are two inputs, or mental spaces, and a third blended space.

Second, because conceptual blending is often considered a "refinement" of the metaphor theory that considers how multiple spaces blend, it allows us to understand what resources are readily available and how they will likely be used to help students make sense of a con-

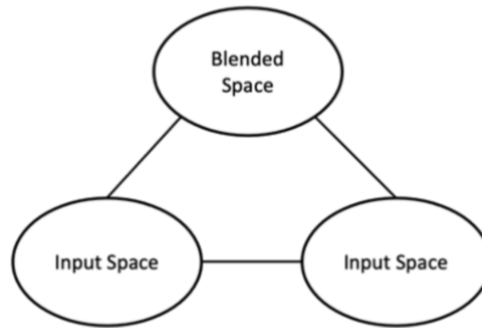


Figure 7.1: Diagram of a Generic Conceptual Blend. There are two input spaces that are "blended" into the third, blended space.

cept. It enables us to understand how multiple resources, embodiment, and metaphors are layered. As Sorva stated, when dealing with computation, one's confusion is about the invisible. Embodiments and metaphors allow one to reify the notional machine [64]. Conceptual blending is then a tool that explains how the embodiment, metaphor, and computation interweave to help students develop understanding and efficacy in solving computational problems. One can see not only the blend but also how it is coordinated.

Third, conceptual blending focuses on the moment-to-moment understandings. Blends are considered generative and creative and yield new ways to think about a source domain. Blends are adaptive, and not just memorized set responses. Surprisingly, only one paper in CER has used conceptual blending as a theoretical framework.

7.3 Methods

In this study, I video recorded student dyads as they collaboratively solve recursive problems. This study is largely qualitative, because I was interested in documenting how embodiment and space supports students problem solving. I present the findings in the style of detailed episodes; this work is meant to be descriptive and interpretive.

Dyadic research has been used extensively in education scholarship to understand how students co-construct knowledge and work collaboratively; a similar style of study in CER

are studies on pair programming [17, 176]. CSEd researchers typically use structured interviews to study the different problem-solving and tracing strategies students use. The crucial difference between individual and dyadic interviews consists of the interaction between participants in dyadic interviews, allowing participants to "co-construct" their version of the topic. Learning is a social process and, thus, should be analyzed as such.

Specifically, I chose to conduct a dyad study because I am working under the assumption that students were more likely to create explanations to each other than with a student-researcher pairing. Explanations are likely to be a rich source of embodied data – gestures, sketches, and language. For example, gestures are communicative, and, therefore, students are more likely to gesture if they are trying to communicate ideas to someone else.

Furthermore, because of the COVID-19 pandemic, data was collected remotely. However, I do not consider the effects of virtual learning. I also do not analyze this data for collaboration, nor did I not setup the study to facilitate collaboration. I only set up the study to facilitate communication.

7.3.1 Participant Recruitment and Compensation

I recruited participants from two undergraduate programming courses, CS 1331 and CS 1332, offered at Georgia Tech via email sent by the course instructor.

The following is the body of the email:

> You are invited to participate in a research project studying how people learn to program. You will be paired with another student of your choosing (both students must currently be enrolled in CS 1332/CS 1331) while you both collaboratively solve some programming problems. If you choose to participate, you will, individually, receive a 25 Amazon gift card. The interview should last no longer than 75-minutes, and it will be video-recorded. The interviews will take place on campus and scheduled at your convenience. If you are interested, please fill out the following form, and a researcher will be in contact.

Please note, we only need 10 students (or 5 pairs).

(link to demographic survey and the pre-knowledge test)

Participating in the study will have no bearing on your standing in the class, and your instructor will not know who participated.

Participants were compensated with a 25 Amazon gift card.

7.3.2 The Course

The courses CS 1331 and CS 1332 were chosen because I wanted to see if students' use of embodied representations might differ based on how much recursion they were taught. I was originally interested in comparing the use of representations between students that were and were not formally introduced to recursion. The purpose was to make claims about how representations transform as one gains knowledge, to see if teachers influence students' use of representations, and hypothesize about the kinds of thinking and reasoning students use at different levels of understanding. We might expect students who did not know much about recursion to create their own representations, while students who know recursion likely used the same representations as teachers. However, I did not find any evidence that representation use differed.

CS 1331 Intro to Object Oriented Programming: CS 1331 is for CS majors and non-majors, and is the second required course for CS majors. The Georgia Tech catalog description of the course describes the course as: "Introduction to techniques and methods of object-oriented programming such [as] encapsulation, inheritance, and polymorphism. Emphasis on software development and individual programming skills." The course uses a typical lab-centric structure with one hour of lecture three times a week and two hours of lab or recitation once a week. In this course, recursion was taught towards the end of the semester. The instructor over this course argued that it is expected that students are introduced to recursion in this course, and will learn it in the next course. This course is taught using the Java programming language.

CS 1332 Data Structures and Algorithms: CS 1332 is the third course for CS majors and minors to take. The Georgia Tech catalog description of the course describes the course as: "Computer data structures and algorithms in the context of object-oriented programming. Focus on software development towards applications." The course uses a typical lab-centric structure with one hour of lecture three times a week and two hours of lab or recitation once a week. In this course, recursion was taught towards the end of the semester. In this course, recursion was reviewed within the first week of class. I should note that the instructor of this course believes that students should know recursion before entering this course and expected them to learn recursion in 1331. This course is taught using the Java programming language.

7.3.3 Participants

As previously stated, participants were students who were currently enrolled in either CS 1331 or CS 1332.

There were a total of 10 dyads (20 students): five dyads of students in CS 1331 and five dyads of students in CS 1332. Students self-selected which partner they wanted to work with. Therefore, all pairs had previously interacted with their partner prior to the study. I required that participants knew each other because I was working under an assumption that pairs that knew each other would be more likely to be more adventurous, talk, etc. This restriction did limit the number of students I could choose from. For example, most racially minoritized and gendered participants that volunteered for the study said they didn't know anyone else to work with in the course. Therefore, I could not include them in the study.

The students that volunteered are likely to be some of the "better" performing students. Moreover, these are students who have consented to learning programming, including the practices. Because of the school they are at, these are also students who likely come from upper-middle class upbringings and who are "good" at school. They likely had high GPAs in high school and high SAT scores.

7.3.4 Data Collection

I emailed interested students a knowledge assessment that included demographic questions about their formal and informal programming experience, the high school and college computing courses they have taken, and their motivation to pursue CS. Demographic questions were at the end of the assessment. There were three multiple choice questions about recursion taken from SCS1, a validated assessment of computer science performance. Because there was a wide variety of experience, I wanted to assess their skill and competency, knowledge to have a sense of what they're willing to do and can do. Furthermore, Parker et al. used item response theory on the SCS1 and have data about past use, so we know if students' performance on the three questions were above or below the average [177].

Sessions lasted between 60 to 90 minutes. Because of the pandemic, sessions were conducted remotely. Each session was conducted using Bluejeans (video communication software) and InVision (an online collaborative whiteboard). BlueJeans allowed students to see each other while solving the problems. During sessions, students were to work with their video cameras and microphones turned on. I turned both of mine off so as not to disturb the students. InVision is an online collaborative whiteboard that allows someone to track and see what someone else is doing in real time. InVision allows someone to draw and type, and tracks ones mouse movements. Therefore, if another student is using their mouse, their partner can see where that student is pointing to. I used this functionality as a proxy for pointing gesture.

I had one computer that was used to screen record the InVision application to capture students while they sketched and to see what they were referring to or pointing to. I used Bluejeans audio recording functionality to capture the video call. Students worked from their own setups.

7.3.5 Dyad Protocol

During the interview, participants talked aloud while collaboratively solving a series of recursion problems. I modeled my protocol after the works of Colleen Lewis [178], who used a grounded approach to study students learning recursion, and Michelene Chi [179] and Jeremey Roschelle [17], who used dyads to understand collaborative learning.

First, I explained the study and answered any questions the participants had. I explained to the participants that I was interested in understanding how they thought about the problem and their process, and wanted them to talk to each other as they solved the problems. I stressed that I was not concerned about the final solution and that getting an answer wrong was not a reflection of them. I also told them this study was not related to their course, would not impact their grades, and the teachers would not know who signed up for the study.

Since most of the dyads had never used InVision, they were given some time to use the software. I had them work on a fun, straightforward exercise that was similar to Picktionary, that gave them the opportunity to get used to drawing and typing in the software.

During the collaborative session, I asked students to help and encourage each other to solve the problems. Participants were presented with an InVision board with all the problems. They were directed when to move on to the next problem. They were asked to read the directions out loud first, and then proceed to solve the problem. The only restriction I gave the students was that they could not use a compiler.

When solving problems, there was one time that I had to prompt students to talk. They remained silent for an extended period of time, I prompted them to continue talking by saying "talk to each about what you're thinking?" are you thinking?" Although most pairs didn't ask me any clarifying questions, some pairs did ask if they were allowed to do something, for example if they could write their own code or if they could change the code however they felt like. My response each time was to do whatever they felt was appropriate.

After a student solved a problem, I asked follow-up questions to attempt to understand

their reasoning better. 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 or asked them to explain their drawings. I also asked students to explain their solutions or to trace the problem and their solutions. The students were told after each problem whether their answer was correct or incorrect.

Originally, I had planned to not tell them whether 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. But, during my first set of participants, I quickly realized that some students in CS need validation otherwise their morale goes down very quickly. One participant kept mentioning how “stupid” or “dumb” they felt and didn’t feel like they could solve any problems correctly, so I started telling students whether or not they got the correct answer just to keep morale alive. I also said other encouraging things during the session like, “you’re doing great”, and other compliments, which seemed to matter for a lot of students and seemed to perk them up while going through all the problems.

7.3.6 Interview Questions

There were three questions (described in more detail in a later section). The first question was a tracing question, the second question asked students to modify a program, and the last question was a Parson’s Problem or a code writing question. I asked both writing and tracing problems because it’s unknown how or if students used embodied representations.

The questions were written in Java, the same programming language in both courses. I created these questions with the help of the CS 1331 and 1332 instructors and teaching assistants. The teaching assistants are undergraduates, but they help put together all coding tests and assignments. I worked with them so to ensure that students would have relevant knowledge to solve the problems, to make sure the instructions to the problems were presented in the language students were familiar with, and to ensure that the problems were

non-trivial to the students in CS 1332, but approachable by the students in CS 1331.

Interview Questions

1. What made you interested in taking this class?
2. Have you taken any other classes, gone to any workshops, or camps that focus on programming or working on your computer at school or other places? How was this class similar or different than those?
3. Let's talk about recursion.
    - How do you define recursion?
    - Why use recursion?
    - Do you like recursion?
    - When would you use recursion?
    - What sort of resources did you use to learn recursion?
    - Where did you learn recursion?

7.4 Data Analysis

7.4.1 Case Selection

I had two streams of video for each dyad: videos for gestures and videos for sketches. I audio transcribed each video. In total, there were 622-minutes of codable data.

I began the analysis by watching the recordings (both the gesture and sketching videos) of each pair while they solved the first problem. I was looking for embodiment, and I used the framework I developed from the professor study as the starting point. I made notes for each pair, including some interesting gestures and utterances, as well as described their sketches and whether they sketched.

Next, I turned that information into an Excel spreadsheet. I wrote categories in the top row of the sheet based on the quotes and high-level points, with the first column comprising each of the pairs. I went through the data and noted whether a pair performed any of the categories. This allowed me to discern any emerging patterns of interest. After some discussion, I went back and did the same exercise for the rest of the problems. I then organized the Excel spreadsheet according to the pairs that got the question right or wrong. This activity highlighted three specific episodes from three dyads and highlighted emerging patterns and behaviors. There was one episode per problem.

Each episode is described in the case studies in later sections. Each episode represents behavior that was out of the norm. The first episode described was the only pair that got that problem wrong. The second episode described included students that used metaphors in intriguing ways compared to the rest of the pairs. The third episode described included a pair who did not follow the typical behaviors of the other pairs that got the problem correct.

For each of the three dyads, I first analyzed the rest of their videos to see if there were commonalities in the ways they solved all the problems. I then described what students needed to know to solve the problem and what was the reason for including that problem. For two of the three problems, I then analyzed four pairs (two who got the problem wrong and two who got the problem right). For the other problem, I selected two pairs who got the problem correct (for this problem only one pair got it wrong). I created multimodal transcripts for each pair. I did this to develop hypotheses to understand the three episodes of interest. It was clear that pairs that got the problem correct displayed similar patterns. Likewise, pairs that got the problem wrong displayed similar patterns.

I then proceeded to code all the video data.

7.4.2 Coding Gestures

I used the methods specified by Trafton et al. [157] and Stieff and Raje [159] to analyze students' gesture production. Using Trafton et al.'s framework, we identified four kinds of

gestures: (1) beat, (2) deictic, (3) iconic, and (4) non-iconic.

Beat gestures were "typically brief, motorically simple gestures" [158], including gestures that go along with rhythmic language, communicative gestures (e.g., thumbs up), and personal gestures (e.g., touching one's nose). Gestures were denoted as deictic if "there was a directed, explicit pointing action, usually involving a finger or forearm, and a purposeful direction toward a display or item in the environment," including pointing to a specific location, thing, or line of code, and typically followed by a demonstrative (this or that). Iconic gestures were any gestures that "acted out" a sequence; specifically, these were gestures "that had a strong relationship to the semantics of the utterance, or 'acted out' what was said." Lastly, non-iconic gestures included gestures that could not be placed into any of the other categories, "were a mix of metaphoric gestures and non-codable gestures (they were not iconics, beats or deictic gestures)" [157]. Some scholars have argued for just collapsing iconic and metaphoric gestures into one category, but we decided to keep them separate. Some of these gestures seemed to "act out" processes or constructs. Knowing a teacher's intentionality, such as asking them what they thought a gesture meant and understanding their underlying metaphors and mental models of constructs and syntax, could help parse it out.

I used a two-step gesture coding scheme. I first reviewed the video recordings with the sound off and tagged every occurrence of a gesture. Following this, we reviewed the recordings a second time with the sound turned on and the concurrent transcripts. During the second viewing, I classified each gesture using the taxonomy previously described.

7.4.3 Metaphor

I used grounded theory to code the transcripts of three videos. I began analysis by open coding, in which I looked for any noteworthy patterns in the data. During this first pass, I coded concepts that were "interesting," particularly words or phrases that we felt "seemed spatial" (e.g., calls, returns, etc.) or any language that seemed anthropomorphic or could

be used to personify.

To determine whether or not the words were metaphors, I used a coding scheme followed by Jeppson et al. and the Pragglejaz Group which provides explicit criteria for categorizing a word as metaphorical [82, 160]. This method has a checklist to determine whether a lexical unit (word or phrase) is a metaphor.

I then performed axial coding on those metaphors. At this stage, I realized that many metaphors were complex and made up of multiple other metaphors. From here, I used conceptual blending as a theoretical framework to explain how the multiple metaphors were combined and layered.

7.4.4 Parsons problems

Parsons problems are typically done with machine learning, similar to path analysis type concerns. As I am focused on sense-making, I wanted to understand students' goals, such as what they were trying to do or what they were trying to make sense of. I began by coding each move five dyads made. For example, I coded when they dragged over each piece and which piece. I then went through and coded where each piece was being dragged into (i.e., inside of an if-statement, outside of an if-statement, etc.). I then performed axial coding to establish what the students were trying to accomplish with each move (e.g. if they were trying to write the base case).

7.4.5 Sketches

When analyzing sketches, I was interested in the process and what students chose to represent, to make inferences about how sketches supported learning. I used these two coding methods. I first coded each time a student drew something and what they drew. I then did axial coding to understand what students were trying to "think with." For the second coding method, I also used an existing taxonomy to categorize each sketch as a way to focus on what information they were conveying [21, 145].

7.4.6 Segmentation

To make claims about what purpose embodiment might serve or the goal behind its use, I conducted a discourse analysis, specifically, a move analysis. I used a discourse analysis methodology to unpack the intent of the students' embodiment. I considered a move to be a distinct shift in focus or a change in topic or purpose.

This adjustment segmented students' conversation into chunks that delimit units of speech produced according to what that speech is doing in the interaction.

Categories of students' moves were developed and refined by two researchers using an iterative process that involved an analysis of the nature and intent of a students' statement. Each move was assigned to an existing code or to a new code as necessary. Where there was disagreement or ambiguity, the researchers discussed this and decided whether the move warranted a new category or pointed to a need for clarification or expansion of meaning within an existing category. The coding categories were refined to the point where all utterances would fit within the coding system.

Table 7.1: Moves and Descriptions

Move	**Description**
reinterpreting the problem	Questioning details of the problem prompt or problem requirements.
searching for analogous example	Identifying similarities between the current problem and other problems or solutions.
proposing a solution	Describing potentional solutions to the problem.
finding issues in solution	Explaining why their proposed solution won't work.
evaluating solution	Judging the correctness of code.
explaining logic	Explaining to partner their understanding of how the code works.
adapting solution	Identifying what needs to change about a prior solution to solve the current problem.
tracing the code	Tracing the code to understand how the code works.

7.4.7 Tradeoffs and Limitations

Students Attending to Embodiment

I was interested in understanding if students attended to their professors embodiment, to begin hypothesizing about textitwhere or the source of some types of embodiment. Past research on sketching has argued that students reject teachers' sketching styles, instead choosing to follow the styles of the teaching assistant. Anecdotally, one professor mentioned to me that they didn't believe students paid attention to her gestures because students had too much to pay attention to in classrooms. However, some of the metaphors and gestures we describe students using are similar to the ones reported in previous chapters on teachers. We need more research on what students attend to in CS classrooms. Before the pandemic, I was observing both the classrooms and recitations to see how the professor and teaching assistants, respectively, taught recursion. However, because of the pandemic, I was no longer able to observe the recitations and classrooms and had to wait a semester to collect data. Therefore, the observation data I did collect was not used in this study. This study was also done completely virtual.

Participants

I had three sets of all-female dyads, three sets of all-male dyads, and two sets of female–male dyads. As previously stated, the dyads were self-chosen. While this created a dynamic that was great for interaction, some obvious toxic masculinity instances arose where the female participants would say something and quickly be dismissed by male participants, or the female participants would consistently ask their male partner if taking a certain action was okay.

Gesture

Because data collection was remote, I was not able to get much nuance about gesture. However, I was able to get the "bigger" gestures, or the salient gestures one makes when they're trying to communicate information. Furthermore, I tried to answer more robust questions about gesture and sketching, such as why would a student choose a gesture over a sketch.

Technology

I used the BlueJeans and inVision applications. Both applications allowed me to see the participant and gave them space to work in collaboration remotely. In BlueJeans, you can choose between three modes, and I chose the mode where the speaker takes up the screen, while the other participants are smaller in the corner. I did this so that I could get a bigger picture of the gesture and could get the student in the frame. The other modes cut off a part of the other persons on screen. The selected mode focuses on the student who is talking, so it is not clear what the other student is doing. However, in the other modes, the image cuts off parts of their bodies. Moreover, inVision allowed space for students to collaborate in real-time. Students could see where the other students' mouse was, but only if they were using a mouse. Some students used a tablet, so the other students could not see where they were pointing.

This made it difficult to analyze pointing gestures since there were some instances where I could not accurately count how often they pointed. Moreover, it did not capture the nuance of big gestures, but the things most salient and what was most viewable came to the top. It must be noted that when you are conscious of it, you make your gesture bigger.

7.5 Findings

In this section, I present the results of my analysis. First, I describe the three interview problems and give an overview of how pairs solved the problems, including what conceptual blends they created, metaphors used, and sketches.

Then, I describe what types of gestures and metaphors appeared consistently throughout all problem-solving sessions.

To contextualize these findings, I conclude by presenting three detailed analysis of one pair solving each problem.

See appendix for code trace for each problem.

7.5.1 Overview of the Problems

Problem 1 - A Tracing Question

Directions: The call mystery(5) returns what value? Write your answer anywhere.

The first question, after the warm-up question, asked students to calculate the value of mystery(5). The invocation to mystery(5) generates an invocation to mystery(4) and multiplies the result of that by 2. This process is repeated and the value of the variable n is repeatedly multiplied by 2. In layman's terms, n is an exponent and this function calculates 2 to the whatever inputted power, n. The correct answer from this set of calculations is 32.

This question was designed to be easy to solve. It was modeled after the recursive example of calculating the factorial of some value n.

Students need to know the following about recursion to solve this problem:

1. Single variable changing (state change)
2. When the recursive process terminates
3. What order recursive invocations execute

```java
public class Problem1 {
    public static void main(String []args){
        System.out.println(mystery(5));
    }
    public static int mystery (int n) {
        if (n == 0) {
            return 1;
        }
        else {
            return 2 * mystery (n - 1);
        }
    }
}
```

Figure 7.2: Problem 1

Pairs Problem Solving. Only one dyad got the first problem wrong. I analyze that dyad in detail in Case Study 1.

All pairs started by tracing the problem. Most pairs realized the function was simply calculating two to the power of whatever value was passed in. Only one pair did not realize that the solution was two to the power of five. They traced the entire solution and got the correct answer, nevertheless.

Every pair drew a sketch. See Appendix for a list of all dyad's sketches for each problem. Two groups created sketches that kept track of how many times the recursive invocation was executed. Since this problem was two raised to the power of five (2 x 2 x 2 x 2 x 2), technically only tracking the number of times the invocation executes would have led

to the right answer. These two pairs also did not realize the answer was two raised to the power of five until after they sketched their traced the code.

The other pairs sketched what would be "in" the stack. That is, they sketched the recursive invocation as an equation, and then went back and filled in the answers to the invocation. Interestingly, the pairs represented the value returned from an execution differently. Some pairs simply used an equal sign with the value next to it. Another pair drew a box and placed the value inside it to represent the execution on pause. I also noted that some pairs used arrows between each invocation. This seemed like a metaphor to hide the highly complex processes happening as well as the processes that were not necessarily important to think about for this answer.

Problem 2 - Modify the Program

Directions: As is, the code produces the following output with the call mystery(2010): 2010 201 20 2

Change the code to get the following output with the call mystery(2010): 2010 201 20 2 20 201 2010

Your solution must be recursive. Write your response anywhere. You don't have to rewrite the code.

This question asked students to change the code to get the new, desired output. As is, the invocation to mystery(2010) checks whether the inputted value n is less than 10. Since it is not, the value n, or 2010, is printed and the computer invokes mystery(201). This process continues until the inputted value is less than 10, at which point the variable n is printed one last time, and the recursive execution terminates.

The solution to this problem is to add a "System.out.println(n)" statement after the recursive statement.

To solve this problem, students need to know about non-tail end recursion. Specifically, students need to know that lines of code after the recursive statement are "held" or "paused"

```java
public class Problem2 {
    public static void main (String[] args) {
        mystery(2010);
    }
    public static void mystery (int n) {
        if (n < 10) {
            System.out.println(n);
        }
        else {
            System.out.println(n);
            mystery(n/10);
        }
    }
}
```

Figure 7.3: Problem 2

until the recursive invocation executes and terminates:

1. Execute outstanding operations (operations after the recursive execution terminates)
2. Stack stores operations

If students do not have that understanding, this problem could be difficult. The code does not make evident that processes can happen after the recursive statements.

Pairs Problem Solving. Three dyads got this problem wrong. The dyads that did get this problem correct, did so fairly quickly.

Interestingly all the groups that got the answer correct mentioned needing to "get back"

```java
public class Problem2 {
    public static void main (String[] args) {
        mystery(2010);
    }
    public static void mystery (int n) {
        if (n < 10) {
            System.out.println(n);
        }
        else {
            System.out.println(n);
            mystery(n/10);
            System.out.println(n)        // <-- The solution
        }
    }
}
```

Figure 7.4: The solution to problem 2 is adding the line in the square.

specific values (or get back something). "Get back" was a metaphor that referenced two important aspects of the stack: (1) the stack stores values and (2) the stack executes statements after the recursive call. When asked how they came up with the solution, they all said they had a similar example and referenced the stack.

Many of the groups started by tracing the code. Two pairs started by looking for patterns in the output and code to see if they had an analogous problem. Both groups then decided to trace the code and eventually solved the problem correctly.

Two pairs sketched a code trace to check that their solution was correct. They drew sketches that were primarily just the output.

The three groups that got the answer wrong used a metaphor that they need to "reach" or "keep" some value. In two of the pairs, one student mentioned they needed to "get

back" values, and described solutions where they put something after the "else" statement. However, the other student in the pair in both dyads quickly dismissed the idea. All their solutions concerned with "storing" values, which involved adding a data structure to "store" or "keep" values and a statement to multiply the inputted value to "get" the value. They did not know or realize that a recursive process typically "stores" values in a stack.

Problem 3 - Parson's Problem (Writing Question)

Directions: Below is an incomplete program. The program is supposed to produce the following output with the call mystery(5, 1):

```
* * * * * #
* * * * #
* * * #
* * #
* #
```

Your job is to complete the mystery method so that the program produces that output.

To the right of these directions is a Test Bank with jumbled lines of code. Drag and rearrange the lines of code below the mystery method to complete the method.

Your solution will include every line of code.

There are brackets in the Test Bank, but you don't have to use them.

Test Bank

```
mystery(n-1, 1);
if (i <= n)
if (n < 1)
System.out.print("*");
System.out.println("#");
mystery(n, i+1);
else
return;
{ } { } { }
```

```
public class Problem3 {
    public static void main (String[] args) {
        mystery(5, 1);
    }

    public static void mystery (int n, int i) {
```

Figure 7.5: Problem 3

This was a Parson's Problem, which is, "a type of code completion practice problem in which the learner must place blocks of mixed up program code in the correct order" (Solving Parsons Problems Versus Fixing and Writing Code). Parson's Problems are considered proxies for writing code problems, but they are generally considered easier to solve.

The invocation to mystery(5, 1) first prints an "*" then generates an invocation to mystery(5, 4). A "*" is repeatedly printed on the same line, until an invocation to mystery(5, 6). A "" is printed on the same line, a new line is created, and then the computer invokes

```java
public class Problem3 {
    public static void main (String[] args) {
        mystery(5, 3);
    }

    public static void mystery (int n, int i) {
        if (n < 1) {
            return;
        }

        if (i <= n) {
            System.out.print("* ");
            mystery(n, i+1);
        }
        else {
            System.out.println("#");
            mystery(n-1, 1);
        }
    }
}
```

Figure 7.6: The solution to Problem 3.

mystery(4,1). This process is repeated until the invocation mystery(1,1).

Students must understand the following to solve this problem:

1. Multiple,concurrent recursive processes
2. Multiple variables changing (state change)
3. When to print
4. When the recursive process terminates
5. What order recursive invocations execute

Pairs Problem Solving. Six pairs got this problem wrong. Two pairs created a sketch to trace their code to check if their solution was correct. sketches.

All the students started by figuring out the base or terminating case. Most pairs suggested they figured out the base case by looking for the "easiest" statement or the one with a value. Interestingly, all pairs initially assumed that the variable "n" was the number of stars and the variable "i" was the number of "s." However, in reality, those variables stood as a counter to keep track of the number of recursive invocations. Students who were unable to solve the problem never made the shift from that understanding of the variables representing the "s" and "*s". Instead, they continuously tried to fix the code under their current understanding. Two pairs created an "else–if" statement and another pair gave up altogether and just started to write their own code. I argue that this problem is likely an example of issues with writing Parsons Problems. This problem is an example of what happens when the metaphor or analogy underlying a program is misaligned with the metaphor or analogy students would choose to write a program. Many dyads said this problem was difficult for them because it was "like reading someone else's code."

I found students used metaphors to talk about the structure of the code (e.g., "closing" or "wrapped" as if it is some encased object). Some dyads even suggested that their programs should "do things inside" a conditional. This metaphor could refer to structurally placing code in the body of a condition or to execute lines of code in the body of a conditional.

7.5.2 Gestures

Beat Gestures

In total, 72.9% of the gestures coded were beat gestures, like a nervous tick. Students touched their hair and face, flicked their hands, etc. They stood completely still while trying to understand what the code was doing or whenever they were confused, evidenced by the confused faces they made.

Deictic (Pointing) Gestures

In total, 3.5% of the gestures coded were deictic gestures. Students used their index fingers or mouse to point to lines of code. They either pointed to lines of code so that the other student would know what they were referring to or trace a function execution. It is likely that number is not representative of how often students used deictic gesture. For example, I could see a student using deictic gesture through the reflection of his eyeglasses, but the gestures were not captured in the recording.

Noniconic Gestures

In total, 22.4% of the gestures coded were noniconic gestures. These types of gestures were made when students made an explanation. These gestures were coded as noniconic because it wasn't clear if they were gesturing the metaphor they used to conceptualize computation or nonsense.

In the next two sections, I describe two types of noniconic gestures that multiple students used.

Stacking Gesture. Students gestured when they (1) defined a stack or (2) explained how a stack operates.

With the first gesture (defined a stack), students would take one hand, palm facing down and move that hand up while stopping in mid-air points. This could be a representation of the stack itself. Pausing on different spots in the air could be the execution frames or different chunks of data in the stack.

With the second, described in Case Study 1, students motioned "popping" elements out of the stack and "pushing" elements into the stack. Recall from the chapter on recursion that stacks have two main functionalities: (1) Pop, that removes an element in the stack, and (2) Push, that adds an element into the stack.

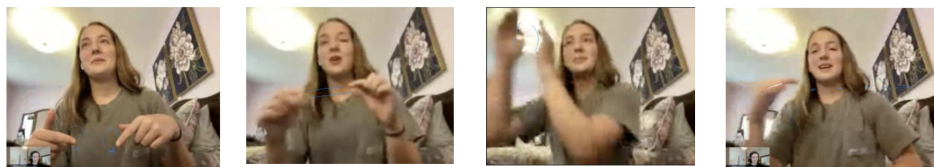
Students would take one hand and motion as if tapping something. This gesture still could emulate the execution frames, but also seems to show elements getting "pushed" or

added into a stack. Moreover, students made a ”popping” out gesture when referring to taking things out of a stack. The students would flick their hand back and forth.

Recursive Gesture. At the beginning of each session, I asked students to independently define recursion. Students made different gestures about recursion based on what aspect of recursion they explained. Recall that recursion has multiple aspects that students need to know: (1) base case, (2) repeating, (3) when it terminates, etc. Some students tried their best to define recursion by including all of those processes.

Most students referred to recursion as repeating itself. Similar to the professors described in previous chapters, those students used an oscillating, circular gesture.

Figure 7.7 is a depiction of Lee explaining recursion to me. When Lee says the line, “I guess whenever a block of code repeats itself”, she makes the oscillating, circular gesture. She then pinches her hands together while dragging them out into a line, while saying, “but it calls it,” and then uses both hands to create an enclosed space while saying, “as like a separate thing,” presumably a separate invocation. Her hand flattens as she moves it from left to right, stopping at random points in space, while saying, “it passes it through again” a literal embodiment of “passing” something through something. Interestingly, the gesture seemed to communicate Lee’s understanding. She seemed to rely on the gestures to communicate when she couldn’t figure out the words to use.



Talk: Yeah. Um, I guess whenever a block of code repeats itself* Action: hands circle around	Talk: but it calls* it Action: pinches right hand, while moving the other hand away	Talk: as like a separate* thing Action: uses hand to form a space	Talk: and it passes* it through again Action: moves hand from left to right

Figure 7.7: Lee uses noniconic gesture to describe recursion. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image.

Metaphors

In this section, I describe the kinds of metaphors students used fluidly throughout the entire session.

Metaphors as An Agent. Students used different pronouns to metaphorically construct ”the computer” (see Table 7.2).

Students would frequently switch between the pronouns, and there did not appear to be any systematicity to it or conscious choices.

In a previous chapter, we described teachers doing this as perspective-taking and role-playing and the pedagogical affordances.

Table 7.2: Different kinds of metaphorical construals.

	Examples
A Computer is an Agent	1. "And so I guess it will keep going into the else clause until it gets to N equals zero." 2. "All right. So, it returns, three returns two times that, then it's going to return two times that, a…"
The Person is the Manipulator	1. "Yeah. Would we just print N again?" 2. "Maybe it's maybe somewhere else we can hold it. We can hold it after. Um, we can do it after a call.…" 3. "The only thing is I don't think you would ever reach that."
Switching Between Perspectives	1. "I think if it does do them zero, then you just get to them one instead of zero. You get what I mean?…" 2. "And it doesn't, so then you look at the else. And the else returns the mystery again, but this time…"

Metaphors to Describe the Base Case and Recursive Process. Similar to Lewis [178], we found two types of metaphors students used to describe the state that satisfies the base case. They describe them as Base-Case-State-is-a-Destination and Base-Case-State-is-a-Goal. In both instances, students use "physical language" to describe the state that satisfies the base case. Students use spatial metaphors that also give a sense of directionality. However, Base-Case-State-is-a-Destination describes the state that satisfies the base

case as if it were a physical location or destination. Base-Case-State-is-a-Goal describes the state that satisfies the base case as a goal. Students use language which suggests that there is intentionality.

We found that students would frequently switch between the two, and there didn't seem to be any systematicity or purposefulness when deciding which to use. They might be unsure of which metaphors are useful in structuring their experience [17].

Moreover, students used the same language to describe a recursive process, including state changes, keeping track of position, connecting to this metaphor of striving. The distinction I make is the metaphors to describe the base case is about describing the goal, and the metaphors to describe recursion is about describing the journey (see Table ???).

Metaphor to Describe a Variable Changing. While tracing an invocation, many dyads used the metaphor "become" when talking about a variable changing, which suggests transformation.

7.6 Case Studies

7.6.1 Case Study 1

This episode emphasizes the following points:

1. students use embodiment and metaphor to construe abstractions using physical notions
2. Using a blend incorrectly

In this 5-minute episode, Aru and Marilyn solve the first problem. Aru is a second-year industrial engineering major, and Marilyn is a second-year computer science major. Both students are in CS 1332.

This episode was chosen for two reasons. First, it is a good example of, generally, dyads' progression through solving the first problem. Second, analyzing this dyad allows investigations into how an exponentiation blend was constructed. Third, they were the only

pair that got the first problem wrong. Therefore, we can also examine how a blend might be productive or ”correct,” but still lead to the wrong answer. Analyzing this pair allows the opportunity to understand the mistakes students might make when relying too heavily on a metaphor or conceptual blend, without really focusing on understanding the code.

Although the pair got this first problem wrong, they demonstrated a good understanding of recursion throughout the session. The pair correctly solved the other two problems, had organized problem-solving strategies, and offered thoughtful explanations for their solutions.

Both students worked on the problems using tablets with styluses and used a separate computer for the video call. Therefore, it was easy for them to create intelligible sketches, but I could not tell if they were pointing to something, what they were pointing at. Aru and Marilyn had their computers in front of them, making their heads through their torsos visible.

After Marilyn reads aloud the instructions to the first problem, Aru and Marilyn sat in silence for 10-seconds. It's unclear what they were doing during this time. They could have traced, looked for patterns, etc.

Aru looks up from her tablet to her computer screen with a slight smile on her face and waits for Marilyn to look at her computer screen. Aru is looking for some signal or embodiment to engage.

Aru constructs a blend to predict when the recursive execution ends: ”And so I guess it will keep going into the else clause until it gets to N equals zero.” The blend contains the following elements:

1. ”it” is the subject, a metaphorical construal of a computer with agency, or the program, or the programmer's intent reified as the program?

2. ”keep going into” is a spatial metaphor that describes directionality, or where the process is going virtually,

3. ”until it gets to N equals zero” is a metaphor Lewis refers to as ”Base-Case-State-is-a-Destination,” where participants describe the state that satisfies the base case as a destination or a location. This metaphor is also a blend containing the elements of a construal of the computer, temporal metaphor, and the code.

4. and the code (else-statement)

Aru’s statement blends to create an elaborated metaphor of the computer’s recursive process as if it is something one has to steer it to stop. This extended metaphor provides a sense that a computer is moving the process forward.

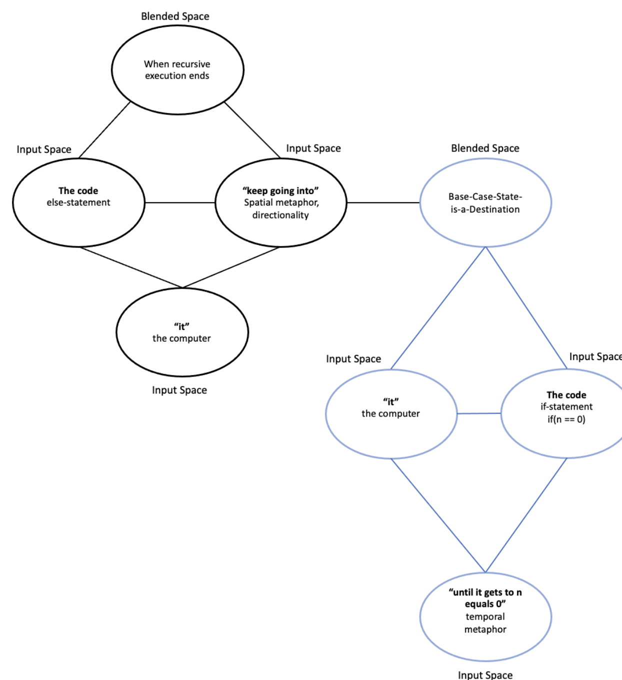


Figure 7.8: Aru’s blend to predict when the recursive process ends. Note, the parts in blue is the base case blend.

Marilyn continues Aru’s point ”Yeah. So, it would be two times mystery zero, and then just keep multiplying by two.”

> Aru: All right. Marylyn, do you want to read them? Marylyn: Yeah. ”The call mystery five returns what value? Write your answer anywhere.” Aru: Cool.

> (silence). And so I guess it will keep going into the else clause until it gets to N equals zero. Marylyn: Yeah. So it would be two times mystery zero, and then just keep multiplying by two. Aru: Yeah. If we write it out like... Ah. Ew. M of four. And then, M of four equals... Oh, yeah. And then... M of... equals two times M of three. Yeah. And then two times M of one. Marylyn: Ah. Okay. Yeah. So this is two, four, eight, 16. Aru: Yeah.

After they decided how the code works, Aru then invents a notional system to illustrate the recursive process. Their sketch as Cunningham states, seems more concerned with the code's functionality rather than the code's behavior. Aru begins by writing 2 * m(4). Aru's writing is in purple, and Marilyn's writing is in red). Interestingly, Marilyn jumps in to complete the sketch and writes 2 * m(5) and an arrow above Aru's sketch. Notice two points. First, Marilyn likely believes she is ”correcting” Aru's mistake by adding an extra recursive invocation and an arrow; the recursive execution does not start at 2 * m(5), but at 2 * m(4) like Aru first wrote. After Marilyn wrote that, Aru agreed that she made a mistake or was wrong by replying ”oh, yeah”. Second, Marilyn finished Aru's initial sketch with no prior conversation during this session about how to draw the sketch. The sketch is a notation that makes sense between the two. There is likely some shared understanding, which could be a shared metaphor or analogy.

They continue drawing each invocation and arrow until the base case executes. Then, starting at the end, Marilyn goes back and writes the output for each execution. She writes an equal sign followed by the computed value. For the last invocation executed, Madeline writes the value ”64” and then ”2^6, ”*suggestingsherealizesthatthisfunctioncalculatestwotothepower*

Another blend happened during this session. This blend that led them to use exponents as an analogy to explain the behavior of the code. This conclusion is evidenced by Madeline writing 2 to the power of 6, and later in the session, Aru says,”Well I thought it was a little counterintuitive. We thought the base case is zero instead of one, because since it's mystery five, you would be like, ”Oh, it's two to the fifth power.” But we're calculating two to the

sixth power, because the base case is zero." when I asked them what was difficult about solving this problem.

The blend consists of the following elements:

1. mathematical knowledge, these students likely have some level of mathematical sophistication

2. the code, specifically the recursive statement and the base case

3. Process - implicit in the blend, but represents the repetition

At the end of the session, I asked them what was difficult about the problem. Aru explains, "Well I thought it was a little counterintuitive. We thought the base case is zero instead of one, because since it's mystery five, you would be like, "Oh, it's two to the fifth power." But we're calculating two to the sixth power, because the base case is zero." This likely might be an incorrect "running" of two blends: the exponent and the termination blend. The execution does stop at "two to the fifth power," but they erroneously believed that the if statement suggests that there is an added recursive invocation.

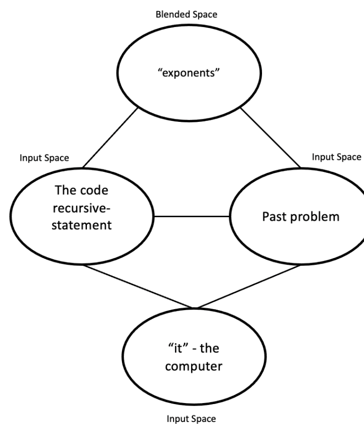


Figure 7.9: Diagram of exponents blend.

I asked the students to explain their sketch and why they chose that representation. Aru's response to the question, "I guess when I've learned about recursion, my teachers

usually talked about a stack, so you keep putting the newer call on top, and then do them from the top down." Aru is likely describing a problem-solving strategy or even a notional machine.

While saying that, Aru takes her right hand, palm facing down and gestures a repeated slapping motion as if something is getting added to the stack. She then uses that same hand and repeatedly flicks it back as if taking away from the stack (see Figure 7.10). Her gestures are a typical depiction of adding items onto the stack and taking items off the stack. This provides even more evidence to the claim that students gesture the metaphor they think about when thinking about abstractions. It is only through the speech-gesture coordination that we can make sense of the notion that the stack pops and pushes elements. Notice, she never actually finishes her statements, but relies on her hand motions to fill in the blanks.

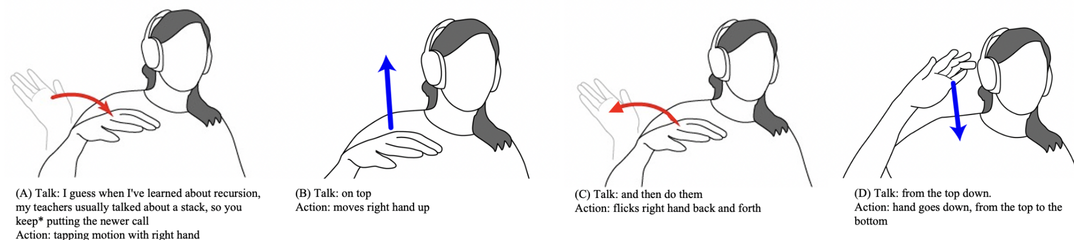


Figure 7.10: Aru's stacking gesture. Talk marked with an asterisk (*) co-occurred with the gestures shown in the image.

This explanation also helps make sense of the process they represented in their sketch. The equation represents each recursive execution, the arrow is a metaphor that hides this complicated process of recursion that they don't need to think about, and the equals sign what happens after each recursive invocation.

7.6.2 Case Study 2

This example highlights the following points:

1. Different blends used while solving a parson's problem

2. Blends added to other blends

3. Using an ”agent” to think through logic

4. Less productive blends

In this 16-minute episode, Audrey and Kartik solve the third problem. They were both second-year CS majors taking CS 1331.

Both students used one computer for both the video call and to solve the problems. Therefore, it was difficult for them to create intelligible sketches, but this configuration did allow me to determine if they were pointing to something and what they were pointing at. Their computer cameras were positioned so that I could see them from their chests and up.

Audrey and Kartik struggled throughout the interview and got the first problem right. Kartik, specifically, rarely finished a statement and frequently interrupted Audrey. At the end of interview, Kartik stated, ”because of the class, I lost hope in my coding skills.”

The students created multiple blends, and I will focus on a few.

Without saying anything, Audrey drags the if (n ¡ 1) statement and asks Kartik to confirm that the statement is the base case. Audrey then drags the return statement and places it under the if (n ¡1) statement. This order of execution checks to see if some value n is less than 1; if it is, the method terminates. Audrey could be determining which statement is the base case by looking for the ”easiest” or simplest statement, a strategy the other dyads admitted to using. The dyads were taught a general rule of thumb for writing a recursive solution is it should have an if-else statement, where the if-statement is the base case, and the base case usually has a number. All the dyads started by assembling the base case. One dyad said, “let’s start with the base case because that’s usually the right thing to do,” which suggests that might have a template or known pattern for writing recursive functions.

Audrey asks Kartik, again, if he thinks what she has put together is correct. At this point, I interrupted them to tell Kartik he muted his microphone.

Kartik suggests that they ”hold onto” the return statement, hinting that he either has

some misunderstandings about the functionality of the base case or that he has misconceptions about return statements.

Kartik drags the mystery(n-1) statement into the base case while saying, ”So, this is obviously the code that's going to work when n minus 1 is there, you get what I'm saying?” Audrey responds, ”I think it's, N is the number of stars... So, if it's less than zero, star is less, I think it should end, don't you think?”

Kartik agrees and suggests that they create an else-statement and move the return statement to the else-statement. Realizing that Kartik doesn't understand her point, Audrey replies, ”Looks like a smaller than sign here. Like this one has as smaller than sign so it's like if there's zero stars left to put, I think it should end, right?” Notice how she constructs an agent (“it”). Frederickson, referencing the seminal work of Ochs and colleagues, refers to the use of pronouns as the “metaphoric construal of pronouns,” but argues that this creates an agent where students can then use to reason with, make predictions, come up with explanations, etc. Similarly, it allows students to use a ”narrative discourse,” that is anthropomorphize abstractions and talk about ”the agent” using everyday experiences.

Audrey has constructed two blends. The first is to understand what the variables n and i represent. This blend consists of the arguments for the invocation, the variables, and the output, to assume that n is the number of stars and i is the number of hashtags. This is a fair assumption this early on. The first row of the output has five stars and one hashtag, and the arguments for the function invocation are five and one.

The second blend is to predict when the base case should terminate. The second blend contains the following elements:

1. the blend about the two variables
2. the code (specifically the conditional)
3. ”it” - the computer
4. ”left to put” - a metaphor for output

Audrey uses the metaphor ”left to put” to reference the output. Notice how Audrey also uses the metaphorical construction of the computer (”it”) as a way to help her reason. Between the blend and ”the agent,” Audrey can talk about the logic of the code using her body.

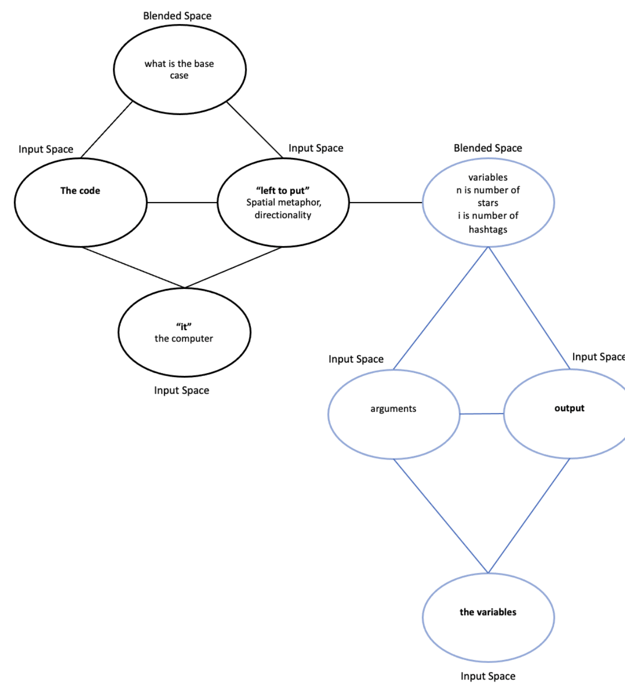


Figure 7.11: Diagram of when the base case should terminate. The diagram in blue is the blend about variables.

After arranging the code for what should happen if (i ¡= n), Kartik suggests they ”do the diagram,” or sketch a code trace. Audrey realizes they still have one statement to place and decides to rearrange the code, moving everything the if-statement inside of the else statement, outside of it, and creates a new else statement. She begins to drag the mystery)i+1) over, before Kartik retorts, “Okay. Hold on. If N is less than one, then that means that there’s-”

Audrey then says, “That’s the last possible thing, I think. So, it’s okay if we don’t have an else and it will return. So, you don’t need an else for that because I think you use the else for the other thing.” Notice, here, that when Audrey ran the ”base case” blend, she

was able to reason and create an explanation for why the else-statement was unnecessary. Notice, here, Audrey switches between “we” and “it.” She uses “we” as if they (humans) own the code and “it” (the computer) as doing the action, or returning. Audrey then uses the pronoun “you”. Who is ”you” here? Kartik? Or is ”you” actually ”we”? Ochs et al. refer to this as an indeterminate construction in which an “extreme form of subjectivity in which the distinction between the scientist as subject and the physical world as object is blurred.” They argue that an indeterminate construction plays an important role in scientists collaboration, by helping scientists achieve a mutual understanding and “arrive at a working consensus.”

Audrey then says, ”Can I put this here? I think after it prints that, it should stop. And then I put this one under this so that it will be done after.” When she says ”Can I put this here,” she drags the return statement back into the if (n ¡ 1) statement. When she says, ”I think after it prints that,” she uses her mouse to point to the ”” in their sketch. Audrey is using the previously defined blend to determine when the recursive execution terminates.

Audrey placed the statement mystery (i + 1) underneath mystery (n - 1).

She constructs a new blend to help her think about where to place a line of code, ”And then I put this one under this so that it will be done after.” In this blend, she layers spatial and temporal metaphors to describe where the recursive process is both physically (in code) and virtually. This blend contains the following elements:

1. ”this one under this” - spatial metaphor about physical location, that is where to physically put the line of code

2. ”will be done after” - a spatial and temporal metaphor; that describes the order in which the execution happens and

3. ”it” - the computer

This seems like a nonsensical statement, but these are all factors Audrey needs to consider when deciding where to put a line of code.

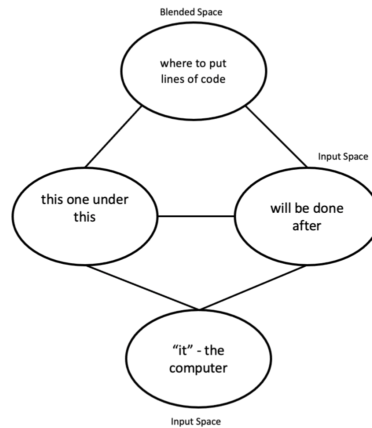


Figure 7.12: Diagram of when to put lines of code blend.

After tracing the code, they can't seem to figure out the purpose of the else-statement. Audrey says, "I think we should move this," and drags the if (n ¡ 1) statement next to the else statement to create an else-if statement. Kartik responds, "Else if ... Oh, that is smart. That is smart. I'll give you that." Audrey responds, "Do we still have the return? Do you think that would make it stop completely? Because I get confused about that." They were never able to understand the purposes of the multiple recursive statements, likely because they did not correctly blend what the different variables represented. Creating an else-if statement seemed to be their way of just getting rid of the else statement.

This pair never got to the right answer. Conceptual blending theory might suggest why, specifically the pair was not able to correctly "blend" their knowledge. The students never understood the purpose of the variables n and i. The entire time, they continued to believe that those variables were for the number of stars and number of hashtags. The blend about what the variables represent did not include one key aspect, that there are two recursive statements that change the values of the variables. Both recursive statements are happening concurrently. The students struggled to understand why both recursive statements were needed. They did, however, try to use the knowledge that everything after the recursive invocation still executes.

7.6.3 Case Study 3

In this example, I highlight the following points:

1. Shifting blends from a less productive blend to a more productive blend
2. Sketching as embodiment and helping to construct a blend
3. Blends help generate solutions

In this 12-minute episode, Vishva and Aniketh solve the second problem. They were both second-year CS majors taking CS 1331.

This episode was chosen because, as previously stated, this pair was the only dyad that took time to solve this problem. Therefore, it allows us to investigate how students try different metaphors or embodied tools until one eventually works.

Both students used the one computer for both the video call and to solve the problems. Therefore, it was difficult for them to create intelligible sketches, but I could mostly tell if they were pointing to something and what they were pointing at. Their computer cameras were positioned so that I could see them from their chests and up.

They begin by mumbling the directions to themselves. Vishva asks, ”what's different?” before answering his question with ”this,” while drawing a line next to the added output. He did not immediately realize the change in output. Aniketh then suggests they start by tracing, ”let's first see how this works.”

Vishva responds, ”Oh, by the way, this gets everything that's to the left. So this is what changes it. This, this.” While saying this, Vishva draws an arrow under the recursive statement to show Aniketh what line he is referencing. Figure ??? is a transcript of the sketch. Visha's writing is in blue, and Aniketh's writing is in green. Vishva continues sketching, writing the invocation's output horizontally, and placing arrows between each outputted value. Like the sketch in Episode 1, the arrow seems to be a metaphor to represent this complex and unknown underlying recursive process. Aniketh jumps in to complete Vishva's sketch, suggesting he agrees that the line of code ”gets everything to the left.”

Vishva justifies that the line ”gets everything to the left,” when he asks Aniketh to remember a problem they had to solve in class. Visvha is searching for a pattern or an analogous problem to make sense of the code, rather than trying to understand the behavior of the code.

Vishva constructed a blend to map between the structure and function. That is, he constructed a blend to make sense of the output of the code. The blend seems to include the following spaces:

1. a past problem
2. ”it” - a metaphorical construal of the computer
3. code, specifically the modulus (%) sign

The resulting blend is a spatial metaphor, ”gets everything to the left,” that could suggest the solution is related to digits or characters in a string.

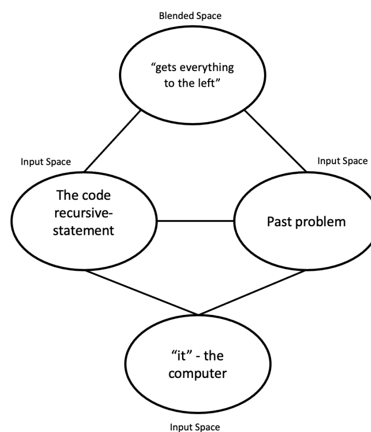


Figure 7.13: Diagram of ”to the left” blend.

As they ”run the blend,” Vishva and Aniketh suggest a potential solution. They suggest to ”store” the value, a metaphor to refer to saving a value so that it can be used later. They continue that they can ”get the values to store” by adding lines of code that multiplies instead of divides a value to get ”the lost value.” When they ”run the blend” we begin to see where their construction ”breaks apart.” The metaphor ”to the left” does focus on

the direction in which a process is happening, which is important to understand to solve this problem. However, this metaphor doesn't highlight that values are already "stored" in a recursive execution. This metaphor ignores the role of the stack, which is to "store" something. There's no frame of reference that those values are already accessible.

Aniketh, clearly not satisfied with any of their options, suggests that they trace the code, a switch in strategies. They create a sketch of the code trace, focusing now on the behavior of the code. Similar to the dyad in Episode 2, they use the metaphor "becoming" to describe a variable changing. Figure ??? is a transcript of the sketch. The first row represents the current value, the second row represents what value the variable n currently is (which is the same as the first row), and the third column represents how the value changes after the recursive invocation executes. Their new sketch is vertical, which follows the conventions of typical stack representations.

While referencing the sketch, Vishva says, "So the main issue is that we can't use recursion to get N back," while drawing an arrow from the bottom-most "n," to the "n" immediately above it. Here Vishva begins to shift blends. When Vishva says, "get N back" he is still using a temporal metaphor, and still referring to storing a value. Aniketh agrees, "Yeah. How do you get this one back? How can you get this number back? How can you get this one back, specifically? Uhmmm, I don't know," while drawing a square around the "1" in "201."

The blend "get N back" includes the spaces:

1. sketch (vertical)
2. The code
3. the computer

However, this blend starts to help Vishva think about the stack. These students were struggling with understanding the output of the code, and because they constructed a blend,

they thought about a metaphor that relates to the stack. This blend helps them talk in typical notions of the stack, that is access this notion of values stored.

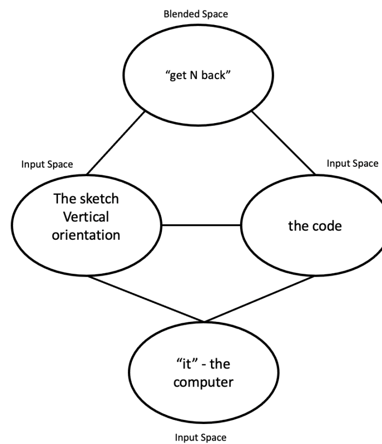


Figure 7.14: Diagram of "get N back" blend.

Vishva suggests that they "should be printing out the things that are lower," where lower could be a metaphor to reference where in the stack some value is placed, or a reference to the actual output and the values after the first "2," suggesting he still hasn't quite grasped at the idea of "getting back." He seems to be using the metaphor "get back" in a temporal sense, as in "getting" a value back. Aniketh responds that they need a return statement.

While Aniketh justifies why they should use a return statement, he comes up with the correct solution, "Wait, what's my number N? No, return N should be it. Because return... Because that's the final case, right? That's the term... I mean, you return to the previous call occurred down here, right? But this thing here, this guy here, oh yeah, so just add a print statement right underneath here. Dude, it's pretty simple I think. You just add a print statement here. And if else. So if you put a return N here, so that you don't keep ... well, actually I don't even think you need a return statement, as a matter of fact. Yeah, you actually don't need a return statement. Just put another print statement right here. That's it."

He gestures a cyclical motion that either conveys an infinite loop or is just to help him get the words out. While he explains, he realizes that they don't actually need a return

statement, and all they need is a print statement.

7.7 Discussion and Implications

Taken together, the three cases and findings presented above expound our understandings of how students make sense of computation and problem solve. The findings indicate that students use different kinds of embodiment to make sense of computation. To close, I reflect on the different sense-making resources students use and the learning implications.

7.7.1 Sense-Making Resources

Metaphors and Conceptual Blends

In this chapter, I used conceptual blending as a theoretical framework, which helped analyze learning and problem-solving as a moment-to-moment, in-situ activity. I found that students seem to be ”fishing” or trying different kinds of embodiment or different blends until they have exhausted all choices or come up with a solution. Students may use a series of pointing gestures to trace the code and then create a blend. They may realize that the blend is not helping them solve the problem, and then they might try sketching a code trace before constructing another blend.

I found that students use metaphors to construe abstractions using physical notions. A base case is something that has to be ”reached,” or a variable ”becomes” a different value. The sheer frequency of metaphors that used physical language suggests that reasoning might be grounded in the physical.

Moreover, I found that students construct blends that often include multiple metaphors. These metaphors represent different abstractions, their past work, the code, and metaphorical agents. These blends produce a metaphor or an elaborated metaphor that helps students reason about, make predictions, and explain the behavior and functionality of code.

Students use a specific type of metaphorical construct, where students granted some indeterminate object (like the computer or the computer process) agency, or the students

were a proxy for the computer. Students talk as if they or some agent were executing lines of code. Jeppsson et al. describe that this metaphorical construal gives students something to ”think with” [82]. The agent helps the student reason about an action (or execution) by priming students to ask questions like, ”what happens next?”

Our findings on metaphors and blends have three implications: (1) programming concepts are likely understood primarily through metaphors or a blend of multiple metaphors, (2) metaphors support reasoning, (3) metaphors provide a language for students to talk about abstractions.

7.7.2 Sketching and Gesture

I also found that students embody code execution when drawing a sketch of a code trace or using their pointer figure to trace.

Sketching provides an “imperfect” model, or a simplified version, of an object or process under study [180]. Students’ sketch the details they want to reason about; they even used symbols such as arrows and equal signs to metaphorically represent more complex problems and situations of process and stack execution. Students used sketching to keep track of program state, drawing how a variable changes or how the elements are popped off the stack. Our findings indicate that the process of sketching is more than just cognitive off-loading [21], but supports students’ reasoning by allowing them to ask questions like “What happens next?”

Although I did not have much gesture data, there is evidence that students used gesture to do a code trace, which means gesture likely supports reasoning. However, whereas a sketch of a code trace can be used to track program state, a gesture of a code trace can only track process; that is where the execution is and where the execution is going. Kirsch argued that gestures are a type of mental image, and mental images hold the same benefits as imperfect models. Gestures are “fast and flexible,” so a gesture of code trace may be more convenient to think with than sketching a code trace. However, a gesture can only

show a limited amount of information.

Lastly, I described instances of gestures that were hard to classify. Students used non-iconic gestures when giving an explanation. The gesture seemed to "embody" or enact certain features about computation. In some instances, the gestures seemed to take on a complimentary representational function, by visualizing aspects students had difficulty talking about. In this sense, these gestures are not only communicative, but seem to be a student's attempt at creating a shared understanding.

7.7.3 Learning Implications

Students Likely Need to Blend Knowledge

Next, we consider the implications of the findings for learning programming. According to Wittman, the conceptual blending framework views the process of learning as combining knowledge from several different mental spaces [173]. A conceptual blend is an activity that leads to knowledge integration, which is the basis for conceptual understanding. Davis describes knowledge integration at its most basic level as "the process of linking scientific ideas together to develop a robust, coherent, conceptual understanding."

In Case Study 2, Audrey and Kartik created less productive blends, likely because they could not correctly blend or integrate their knowledge (table reference here). Consequently, it suggests that blends are more productive for sense-making when the blend does work by connecting disparate spaces. That is, a blend is less productive when it seems unlikely to promote knowledge integration processes and "more productive" when it is more likely to foster knowledge integration processes.

Silvis argued that coding should be construed as conceptual blending [175]. Silvis's framing does a better job of considering the messiness and creativity of problem-solving and sense-making in learning to program. For example, it seems surprising that students would overlay multiple metaphors (e.g., Blend A, B, etc.). However, trying to figure out where to put a line of code or when the base case executes are dubiously simple. In reality,

students have to consider multiple implications. To figure out where to put a line of code, students have to consider where to physically put a line of code and how that choice affects the recursive process. The result is a blend - like Blend C - that seems nonsensical but is a clever attempt by students to handle these disparate spaces.

Accessing Knowledge

Our findings suggest different ways to help students access knowledge they already have so that they can apply it. Vishva's and Aniketh's problem-solving session in Case Study 3 suggests that the process of blending is complex, and students blend their knowledge in many different ways. Aniketh and Vishva constructed different blends to try to come up with solutions about accessing specific information or data. These students were eventually able to get the problem correct, suggesting they had the required knowledge, but were not able to initially blend their knowledge in productive ways. They were struggling to understand the output of the code, and then they created these blends that help them realize they do in fact know about stacks. They thought about that metaphor because they constructed a blend. Blending theory suggests that these students' difficulties do not seem to be not necessarily from a lack of knowledge, but rather from less productive blending of the knowledge of past examples, the structure of the code, etc.

Challenges with Learning to Program

Our findings suggest another reason why learning to program is such a complex and difficult activity. We found students construct multiple blends to make sense of different parts of computation. For example, in Case Study 2, students constructed a blend to predict when the recursive invocation stops, what different variables represent, what an if-statement might mean, the behavior of the code, etc. Moreover, some blends become parts of larger, more complicated blends. Students need to manage the multiple metaphors and blends, and we should consider ways to facilitate or to help students manage.

CHAPTER 8
CONCLUSION AND IMPLICATIONS

I started this investigation motivated by the recent findings in CER that there are correlations between spatial ability assessments and CS performance. While the correlations may be due to several reasons, the implications that, in CS education, we use space to make sense of and learn about abstractions was intriguing. Inspired by Gauvain [18] and Kafai [19], I argued that CER needed theoretical pluralism to understand how space influences CS learning. Theoretical pluralism does not suggest that one theoretical perspective is better than the other but that only looking at one is incomplete: it doesn't give us a complete understanding of how phenomena operate within a context. Typically, CER has treated learning as a solo activity, pulling heavily from only cognitivist learning theories. In this dissertation, I treated learning as distributed, social, and communicative. This framing structured how I investigated space in CS learning.

This thesis was naturalistic and inductive because little was known about how students use space to support CS learning. Inspired by research in the learning and cognitive sciences, I used qualitative methods to investigate CS learning through two main avenues: (1) how teachers use space while explicating computation and (2) how students use space to problem solve. Consistent with research in STEM education - specifically science, mathematics, and physics - I found a close tie with embodiment and space. Specifically, teachers and students use embodiment to create representations of abstract concepts.

In the first study (Chapter 5), I used grounded theory to analyze a set of naturalistic video recordings of undergraduate computing professors teaching recursion to their class. While looking for space, I found an interesting relationship between how teachers use the body and their body-based experiences to help students understand recursion. However, CER has yet to explore how instructors use embodied actions and ideas when teaching the

skills involved in ”doing and learning” computing. For example, I described an instance of a professor using a series of gestures to ”act out” adding elements to a data structure. I also described how a teacher used sketching, perspective-taking, and spatial metaphor to conduct a code trace. I contributed a conceptual framework of the kinds of embodied representations teachers use in computing classrooms as the first step towards understanding how embodiment supports student learning.

In the second study (Chapter 6), I used the same data corpus from the first study to understand how teachers communicate about computation and how well students can interpret it. In this study, I found that metaphor likely plays a central role in CS learning and new challenges that likely contribute to making learning to program difficult. Specifically, metaphors are how teachers conceptualize computation; they use metaphors to describe abstractions, create metaphorical construals of agents, and their gestures about computation seem to be based on a metaphor. I argued that students need to interpret this embodiment since it is one of the few resources teachers are likely using to make computation and the notional machine physical. Teachers used gestures that could be about computation, but, as I argued, students could only interpret those gestures if they had similar metaphors. Moreover, professors explained that recursion is ”when a function calls itself,” layering multiple metaphors and acting as if a function has agency and human behaviors. Professors traced code with utterances like ”then I go here and check if it's less than 1,” which requires students to reconcile with code as not just lines of text on a screen. Lastly, professors use perspective-taking, which asks students to role-play as different characters. However, professors switch between perspectives with no systematicity, and they never make it clear what role the student should be playing. I concluded that the embodiment professors used is not a product of intentional design, and CER needs to think about intentionally designing embodiment for explanatory power.

In the final study (Chapter 7), I video recorded 10 student dyads while they solved recursive problems to describe how students use embodiment while problem solving to make

sense of computation. I used conceptual blending as a theoretical framework. Students create these complex statements that overlay multiple metaphors to make predictions or explanations. Each of these metaphors is about different abstractions. While these statements seem complex, students have to blend multiple metaphors to think about computation. For example, students created this elaborated blend about when the base case terminates, including metaphors about the recursive process, when the process stops, and the condition for stopping. However, those are all factors students need to consider when thinking about when the recursive process ends.

8.1 Implications

8.1.1 Learning Implications

Across all three studies, I found different kinds of metaphor:

1. Metaphors like "calls" and "runs" might be categorized as dead metaphors. A metaphor has become so widely used and accepted, that it's original roots or relationship to its original metaphor is "dead" and has become standard language.

2. There are metaphors that result from a conceptual blend that are used in a specific context, at a specific moment to help someone reason and predict about the behavior of code.

3. There are metaphors that use physical language to describe abstractions (e.g., "it goes here" and "variable n is bound to 0").

4. There are metaphorical - and indeterminate - construals of the computer.

5. Lastly, there are metaphors that seem to underlie gestures about computation. It seems that when someone gestures, they are gesturing their underlying metaphor.

CER has typically only considered metaphors to be a pedagogical tool. However, Watt [126] and Manches et al. [80] argue that embodiment offers a different perspec-

tive on metaphors in computing cognition. Rather than just making abstract concepts more tractable, metaphors may be important in the way certain computing concepts are conceptualized. That is, metaphors are likely how we make sense of, reason about, and communicate computing concepts (similar to claims in mathematical cognition).

Their assertions, however, were only hypothetical. In my dissertation, I inductively found a relationship between metaphor, space, and CS learning. Teachers and students use metaphors to communicate and understand computation. Teachers describe function invocation as if a function can "call" something. Moreover, students talk about variables changing as if a variable can "become" a new value. Students and teachers talk about computation using physicality.

In summation, CS learning is layered with metaphors. Metaphors might be how computation is conceptualized, how computation is described, and one of ways we make sense of computation.

8.1.2 Conceptual Framework

The conceptual framework used to analyze data was based on the grounded analysis conducted in the first study (Chapter 5).

After conducting the third study (Chapter 7) and using conceptual blending as a theoretical framework to explain students' use of metaphor, teacher's use of some metaphors might be better represented as a conceptual blend. Consider when a student asks a question for clarification, the metaphors teachers use as a response are likely a conceptual blend. Conceptual blend more accurately accounts for the ad-hoc, impromptu metaphors or explanations teachers create.

The conceptual framework should be refined and iterated upon by analyzing different contexts. Specifically, teacher-student contexts, whole classrooms that consider both the teacher and the students, or even student and teaching assistant (TA) dynamics - it is common for students to practice coding in separate labs outside of class time ran by the TA.

Regardless, this conceptual framework presents the first to try to attempt to link space, embodiment, and computation.

8.1.3 Pedagogical Implications

As previously stated, Dijkstra and other CEd researchers have argued against the use of embodiment and metaphor. Dijkstra considered it childish to compare computational processes to physical things instead of understanding them for what they are (i.e., "the radical novelty") [9].

However, I found that embodiment and metaphor are critical parts of CS learning, and by ignoring them, we missed the opportunity to design our metaphors and embodiment for ease of understanding and explanatory power. We ought to correct this mistake and embrace the design and use of metaphors in CS learning and teaching. These are similar arguments made by CEd researchers who study culturally-relevant pedagogy and non-native English speakers. They argue that programming syntax and languages follow western epistemologies and likely contribute to people in places like the Global South not succeeding in CS.

Previous work in STEM education has demonstrated that using different metaphors can lead people to reason differently about notions such as energy, time, emotion, or electricity [86, 82]. These findings are based on a body of work that argued for and found the importance of linguistic framing in reasoning. For example, in science education, many researchers who focus on developing learners' conceptual understanding of energy have emphasized the advantages of using a substance metaphor for energy. Conceptualizing energy as a substance is helpful because it tacitly brings along many useful properties in terms of accounting for the transfer and conservation of energy.

CS learning should consider that level of thoughtfulness when using metaphors.

8.2 Contributions

This dissertation provides some of the first empirical evidence on how space is used in CS learning. This dissertation applied learning theories and used methods that had not previously been applied to computer science education. Through this application, I also extend the learning theories to the domain of computer science. The outcomes of this research make theoretical, methodological, and practical contributions.

1. Documenting how space appears in CS learning and using the embodiment as a theoretical framework. This dissertation presents one of the first studies in computing education to describe how students and teachers use space and embodiment to describe the different kinds of embodiment students use. Moreover, while sketching has been studied in CER, this is one of the first studies to consider sketching as a type of embodiment, which frames sketching as "thinking through action." But doing so allowed us to analyze the things that students are likely "thinking" to make sense of computation.

2. A better understanding of how students learn computation and what resources support that learning. While some research has studied metaphor and analogy, it has only thought about them as a pedagogy. In this dissertation, I provide evidence that metaphors are likely central to students' understanding of computation. Moreover, I describe different kinds of metaphors, from the ones students construct, to the ways the discipline has described them.

3. Introduced conceptual blending to CER. I present one of the first studies in CER to use conceptual blending as a theoretical framework. As such, I was able to identify some of the sense-making practices and resources students use to understand computation. Moreover, I presented a diagramming style researchers in CEd can use to describe the relations amongst different elements.

4. New considerations for what makes learning computing hard. As previously stated, CER has not considered the contributions of the body for learning and communicating knowledge. While studying teachers, I found teachers used complex and layered embodiment that likely requires similar metaphors to make sense of it. Moreover, in the final study with students, I found students create multiple blends to make sense of and make predictions about concepts like (e.g., when does the base case terminate, what do variables represent). These blends are complex and students have to manage multiple blends.

5. Methodological. I present one of the first dyad studies in CER.This is a valuable contribution to our understanding of what is learned in CS classrooms and how that learning happens. Moreover, this analysis of learning can provide a template for the systematic study of types of thinking and learning in CS such as spatial thinking.

8.3 Future Work

The contributions of this work have opened significant future pathways for continued research and exploration.

1. **How might the methods or findings be adapted to investigate spatial thinking in CS learning?** As previously mentioned, I was initially intrigued by the correlations between spatial skills assessments and CS performance. In other disciplines, researchers used qualitative methods to study this phenomenon and examined the representations made available to learners.

2. **In-situ analyses.** Although I analyzed videos of CS teachers teaching in classrooms, I did not analyze what students were doing in those settings. Collecting data in labs or recitations where students are taught by teaching assistants would provide richer data about how embodiment appears and influences CS learning. This could eventually lead to taxonomies and more specific design interventions.

3. **Teachers' Intentionality and What do Students Attend to.** In my dissertation, I make many claims about how and why teachers use embodiment and its importance for learning. However, these are just hypotheses. Future work should consider teachers' intentionalities and what students attend to. This research might continue video recording professors, but then ask them to explain what they meant by using this metaphor, explain this gesture. Moreover, future work should consider the source of students' gestures and metaphor use. In the final study, students used similar gestures and metaphors to the ones teachers used. However, some past research has argued that students don't attend to what teachers are doing.

4. **How does embodiment change using different notional machines of recursion or just studying different programming concepts?** This study only looks at embodiment using recursion and stack-based notional machine. How might embodiment change looking at different computing constructs or even looking at different notional machines of recursion? With this, we can have a better understanding of which types of embodiment are better at facilitating information.

5. **Sociocultural context.** My dissertation uses embodiment as a theoretical framework to explore how students learn. Central to my argument is that learning in CS is a joint sense-making process mediated by communicative practices, including embodied representations. Issues of equity and social justice are implicit and explicit for understanding how classroom communicative practices impact CS learning. CS classrooms are cultural and social spaces. Therefore, social inequities are easily perpetuated by using communicative practices that privilege students with certain forms of knowledge.

Appendices

APPENDIX A

DEMOGRAPHIC SURVEY

This survey will ask you questions about your demographic background and three programming questions.

First Name

Last Name

Preferred Email Address

Consider the following code segment.

```
DEFINE iD (x, y)
     RETURN iDCount(x, y, 0)
ENDDEF

DEFINE iDCount (x, y, count)
     IF x <= y THEN
          RETURN count
     ELSE
          RETURN iDCount(x-y, y, count+1)
     ENDIF
ENDDEF
```

Which of the following statements is true after calling the function below?

`iD (17, 5)`

- The value of x is never less than or equal to y; error generated, `iDCount` is an undefined identifier.
- The value of x is never less than or equal to y, so the final answer is `iDCount(13, 5, 1)`.
- The value of x reached or is less than y, so the computer returned 2.
- The value of x reached or is less than y, the computer returned 3.
- The value of x reached or is less than y, the computer returned 5.

```
DEFINE reverse(s)

     IF  (*/missing code 1*/) THEN
          */missing code 2*/
     ELSE
          PRINT s[length(s) - 1]
          */missing code 3*/
     ENDIF
ENDDEF
```

Write code to fill in `/* missing code 1 */`, `/* missing code 2 */`, and `/* missing code 3 */` to complete the mirror function such that it prints the mirror image of a valid string s, where s is a non-empty string.
For example the function call `reverse("Computers")` would print

```
'sretupmoC'
```

```
/* missing code 1 */ /* missing code 2 */ /* missing code 3 */
```

```
/* missing code 1 */ length(s) == 0
/* missing code 2 */ RETURN ""
/* missing code 3 */ RETURN s[length(s) - 1] +
                        reverse(s.substring(0 : length(s) - 1))
```

```
/* missing code 1 */ length(s) == 1
/* missing code 2 */ PRINT s[0]
/* missing code 3 */ reverse(s.substring(0 : length(s) - 1))
```

```
/* missing code 1 */ length(s) == 0
/* missing code 2 */ PRINT s[0]
/* missing code 3 */ reverse(s.substring(0 : length(s) - 1))
```

```
/* missing code 1 */ length(s) == 0
/* missing code 2 */ RETURN s[0]
/* missing code 3 */ RETURN s[length(s) - 1] +
                        reverse(s.substring(0 : length(s) - 1))
```

```
/* missing code 1 */ length(s) == 1
/* missing code 2 */ RETURN ""
/* missing code 3 */ reverse(s.substring(1 : length(s) - 1))
```

```
DEFINE juiceple (s, c, value)

      IF length(s) == 0 THEN
            RETURN value
      ELSE IF s[0] == c THEN
            value = value * 3
            RETURN juiceple(s.substring[2:length(s)],c,value)
      ELSE
            value = value + 2
            RETURN juiceple(s.substring[1:length(s)],c,value)
      ENDIF

ENDDEF
```

What is the value returned by the function call below?

`juiceple("Tennessee", "e", 0)`

- 0
- 18
- 28
- 78
- 306

What is your current academic year?

- 1st year
- 2nd year
- 3rd year
- 4th year
- 5+ year

If you took high school computing, which computing course(s) did you take?

Please list the computing courses you have taken while at Georgia Tech.

Please respond to the following statement: I have a lot of self-confidence when it comes to programming.

- Strongly agree
- Somewhat agree
- Neither agree nor disagree
- Somewhat disagree
- Strongly disagree

Age

- Under 18
- 18 - 24
- 25 - 34
- 35 - 44
- 45 - 54
- 55 - 64
- 65 - 74
- 75 - 84
- 85 or older

Gender (please leave blank if prefer not to answer)

Race or Ethnicity (please leave blank if prefer not to answer)

Does your mother work in a computing field?

- Yes
- No

Does your father work in a computing field?

- Yes
- No

What was your mother's highest level of education?

- no formal education
- elementary (1 - 5)
- middle school (6 - 8)
- high school (9 - 12)/GED
- Bachelor's degree
- Master's degree
- Advanced graduate work or Ph.D.
- not sure/choose not to answer

What was your father's highest level of education?

- no formal education
- elementary (1 - 5)
- middle school (6 - 8)
- high school (9 - 12)/GED
- Bachelor's degree
- Master's degree
- Advanced graduate work or Ph.D.
- not sure/choose not to answer

Do you own a touchscreen device or a device you can use with a stylus that is not a smartphone (e.g., iPad, touchscreen computer, tablet, etc.)?

- Yes
- No

Please write down the names of any students you would like to work with in this study. The other student must also currently be enrolled in CS 1332.

Please list some general days of the week and times that work best for you

APPENDIX B

DYAD STUDENT SKETCHES

Paige and Ritvik's Sketches

Problem 1

32

Problem 2

```
public static void mystery (int n) {
    if (n < 10) {
        System.out.println(n);
    }
    else {
        System.out.println(n);
        mystery(n/10);
    }
}
```

Problem 3

Aru and Maryln's Sketches

Problem 1

your answer

2 * m(5) = 64

2 * m(4) = 32

2 * m(3) = 16

2 * m(2) = 8

2 * m(1) = 4

2 * m(0) = 2

Problem 2

Blank Page

Problem 3

Tanishq and Muhammed's Sketches

Problem 1

```
}
public static int mystery (int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return 2 * mystery (n - 1);
    }
}
```

2 · m(4)

2 * m(3)

2 · m(2)

2 · m(1)

2 · m(0)

2^5

"32"

Problem 2

```
if (n < 10) {
    System.out.println(n);
}
else {
    System.out.println(n);
    mystery(n/10);
}
```

SOP(n);

Problem 3

Blank Page

Khalaya and TJ's Sketches

Problem 1

Problem 2

Blank Page

Problem 3

Charlie and Riley's Sketches

Problem 1

Problem 2

Problem 3

Maya and Ruchi's Sketches

Problem 1

Problem 2

Blank Page

Problem 3

Blank Page

Lee and Caitlin's Sketches

Problem 1

Problem 2

Problem 3

Audrey and Kartik's Sketches

Problem 1

Problem 2

Blank Page

Problem 3

Hayden and Griffin's Sketches

Problem 1

Problem 2

Problem 3

Aniketh and Vishva's Sketches

Problem 1

Problem 2

Problem 3

APPENDIX C

CODE TRACE FOR EACH PROBLEM

Figure C.1: Code trace for problem 1.

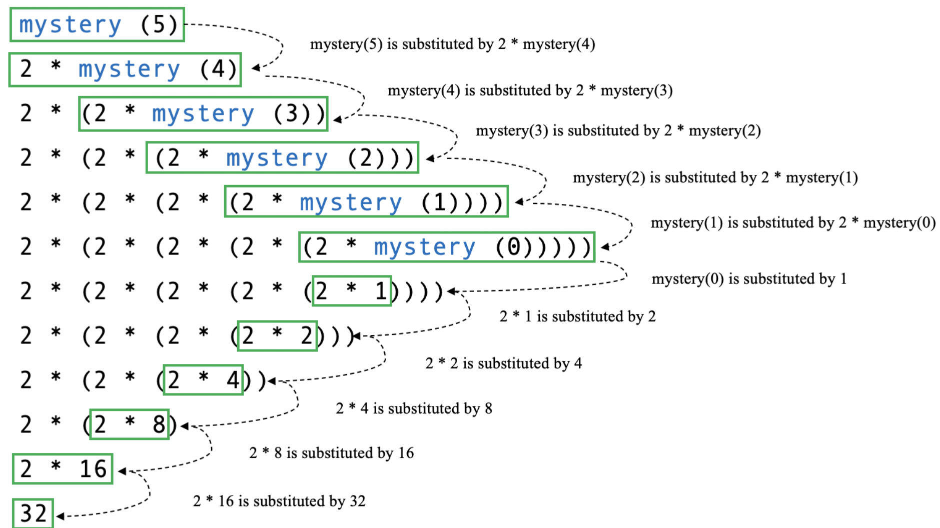


Figure C.2: Code trace for problem 2.

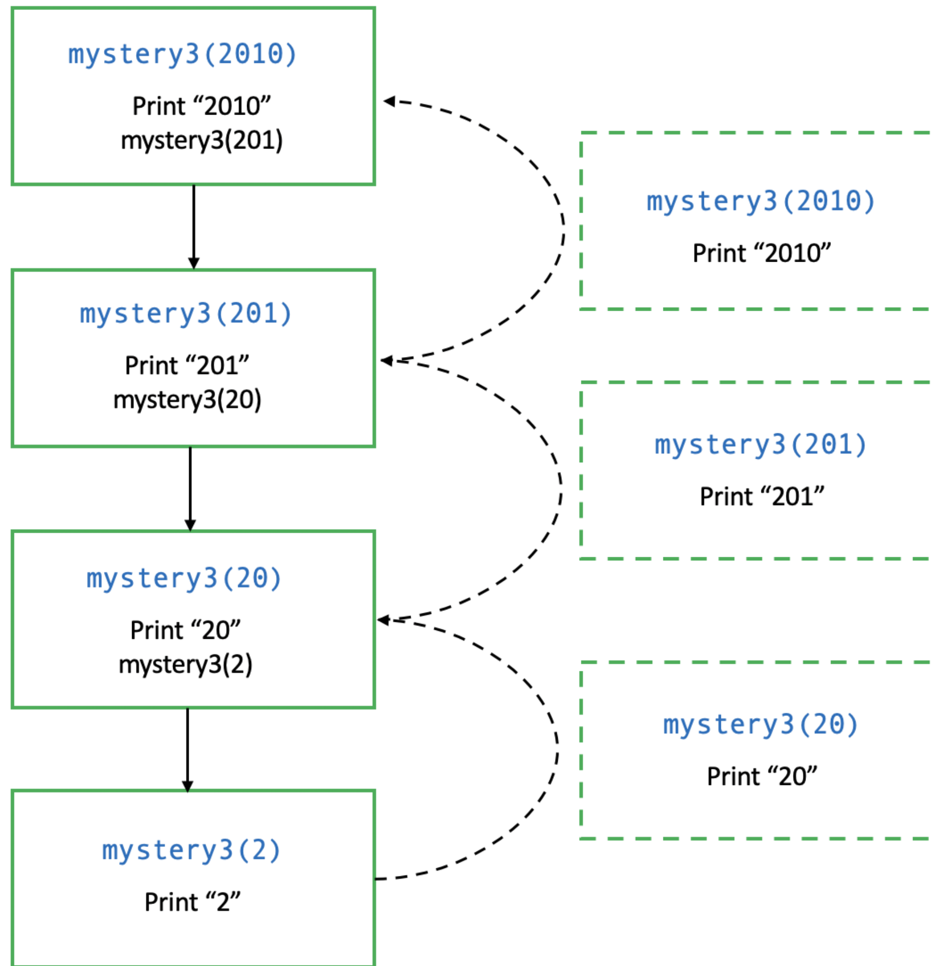
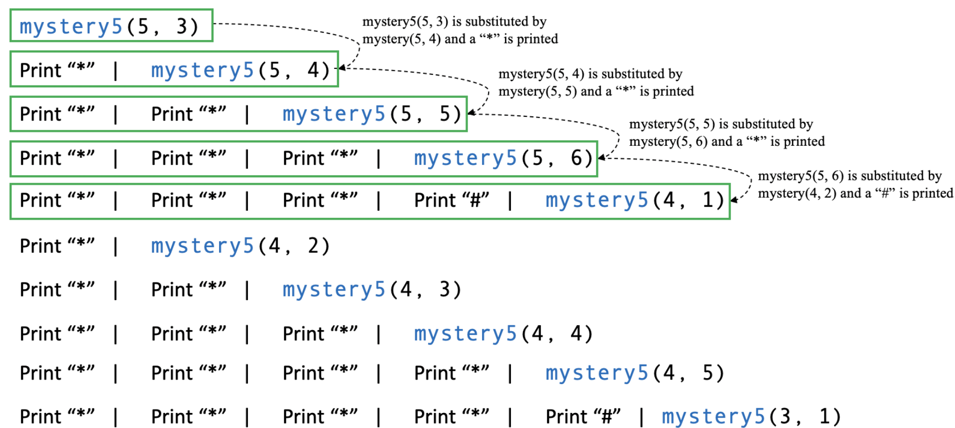


Figure C.3: Code trace for problem 3.



REFERENCES

[1] S. Cooper, K. Wang, M. Israni, and S. Sorby, “Spatial skills training in introductory computing,” in *Proceedings of the eleventh annual international conference on international computing education research*, 2015, pp. 13–20.

[2] M. C. Parker, A. Solomon, B. Pritchett, D. A. Illingworth, L. E. Marguilieux, and M. Guzdial, “Socioeconomic status and computer science achievement: Spatial ability as a mediating variable in a novel model of understanding,” in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, 2018, pp. 97–105.

[3] J. Parkinson and Q. Cutts, “Investigating the relationship between spatial skills and computer science,” in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, 2018, pp. 106–114.

[4] L. E. Margulieux, “Spatial encoding strategy theory: The relationship between spatial skill and stem achievement,” in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, 2019, pp. 81–90.

[5] M. Kozhevnikov, M. A. Motes, and M. Hegarty, “Spatial visualization in physics problem solving,” *Cognitive science*, vol. 31, no. 4, pp. 549–579, 2007.

[6] R. Stevens, “The missing bodies of mathematical thinking and learning have been found,” *Journal of the Learning Sciences*, vol. 21, no. 2, pp. 337–346, 2012.

[7] T. J. Cortina, “Reaching a broader population of students through” unplugged” activities,” *Communications of the ACM*, vol. 58, no. 3, pp. 25–27, 2015.

[8] J. P. Sanford, A. Tietz, S. Farooq, S. Guyer, and R. B. Shapiro, “Metaphors we teach by,” in *Proceedings of the 45th ACM technical symposium on Computer science education*, ACM, 2014, pp. 585–590.

[9] E. W. Dijkstra *et al.*, “On the cruelty of really teaching computing science,” *Communications of the ACM*, vol. 32, no. 12, pp. 1398–1404, 1989.

[10] Y. Cao, L. Porter, and D. Zingaro, “Examining the value of analogies in introductory computing,” in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 2016, pp. 231–239.

[11] N. S. Newcombe, "Picture this: Increasing math and science learning by improving spatial thinking.," *American Educator*, vol. 34, no. 2, p. 29, 2010.

[12] R. Bockmon, S. Cooper, W. Koperski, J. Gratch, S. Sorby, and N. Carolina, "A CS1 Spatial Skills Intervention and the Impact on Introductory Programming Abilities," no. 3, pp. 766–772, 2020.

[13] D. H. Uttal and C. A. Cohen, "Spatial thinking and stem education: When, why, and how?" In *Psychology of learning and motivation*, vol. 57, Elsevier, 2012, pp. 147–181.

[14] H. A. Simon, *The sciences of the artificial*. MIT press, 2019.

[15] S. M. Weisberg and N. S. Newcombe, *Embodied cognition and stem learning: Overview of a topical collection in cr: Pi*, 2017.

[16] N. Enyedy, J. Danish, and D. DeLiema, "Constructing and deconstructing materially-anchored conceptual blends in an augmented reality collaborative learning environment," 2013.

[17] J. Roschelle, "Learning by collaborating: Convergent conceptual change," *The journal of the learning sciences*, vol. 2, no. 3, pp. 235–276, 1992.

[18] M. Gauvain, "The development of spatial thinking in everyday activity," *Developmental Review*, vol. 13, no. 1, pp. 92–121, 1993.

[19] Y. Kafai, C. Proctor, and D. Lui, "From theory bias to theory dialogue: Embracing cognitive, situated, and critical framings of computational thinking in k-12 cs education," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, 2019, pp. 101–109.

[20] A. J. Ko, *Dagstuhl trip report: Learning and teaching programming language semantics*, Jul. 2019.

[21] K. Cunningham, S. Blanchard, B. Ericson, and M. Guzdial, "Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, 2017, pp. 164–172.

[22] P. Tunnell Wilson, K. Fisler, and S. Krishnamurthi, "Evaluating the tracing of recursion in the substitution notional machine," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 1023–1028.

[23] G. L. Herman and N. G. Ave, “The Affordances and Constraints of Diagrams on Students ’ Reasoning about State Machines,” pp. 173–181, 2017.

[24] A. Solomon, V. Oguamanam, M. Guzdial, and B. DiSalvo, “Making cs learning visible: Case studies on how visibility of student work supports a community of learners in cs classrooms,” in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 2019, pp. 161–167.

[25] A. Solomon, M. Guzdial, B. DiSalvo, and B. R. Shapiro, “Applying a gesture taxonomy to introductory computing concepts,” in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, 2018, pp. 250–257.

[26] S. Ainsworth, “DeFT : A conceptual framework for learning with multiple representations . Learning and Instruction , 16 , 183-198 This is a prepublication version of Ainsworth , S . (2006). Deft : A conceptual framework for DeFT : A Conceptual Framework For Considering Learning with Multiple Representations,” no. April, pp. 183–198, 2020.

[27] K. D. Gutiérrez, T. Sengupta-Irving, and J. Dieckmann, “Developing a mathematical vision,” *Language and mathematics education: Multiple perspectives and directions for research*, pp. 29–71, 2010.

[28] M. W. Alibali and S. GoldinMeadow, “Gesture-speech mismatch and mechanisms of learning: What the hands reveal about a child’s state of mind,” *Cognitive psychology*, vol. 25, no. 4, pp. 468–523, 1993.

[29] L. Liben, “Embodiment and children’s understanding of the real and represented world,” *Developmental perspectives on embodiment and consciousness*, pp. 191–224, 2008.

[30] L. Liben, “Representational development and the embodied mind’s eye,” *Body in mind, mind in body: Developmental perspectives on embodiment and consciousness*, pp. 191–224, 2008.

[31] K. E. Ramey and D. H. Uttal, “Making sense of space: Distributed spatial sensemaking in a middle school summer engineering camp,” *Journal of the Learning Sciences*, vol. 26, no. 2, pp. 277–319, 2017.

[32] K. E. Ramey, R. Stevens, and D. H. Uttal, “Steam learning in an in-school makerspace: The role of distributed spatial sensemaking.,” in *Proceedings of the 13th International Conference of the Learning Sciences, London, UK*, vol. 1, 2018.

[33] S. Vossoughi, M. Escudé, F. Kong, and P. Hooper, “Tinkering, learning & equity in the after-school setting,” in *annual FabLearn conference. Palo Alto, CA: Stanford University*, 2013.

[34] R. Stevens, “Learning as a members’ phenomenon: Toward an ethnographically adequate science of learning,” *Yearbook of the National Society for the Study of Education*, vol. 109, no. 1, pp. 82–97, 2010.

[35] R. R. Stevens, “Divisions of labor in school and in the workplace: Comparing computer and paper-supported activities across settings,” *The Journal of the Learning Sciences*, vol. 9, no. 4, pp. 373–401, 2000.

[36] S. C. Levine, M. Vasilyeva, S. F. Lourenco, N. S. Newcombe, and J. Huttenlocher, “Socioeconomic status modifies the sex difference in spatial skill,” *Psychological science*, vol. 16, no. 11, pp. 841–845, 2005.

[37] M. C. Linn and A. C. Petersen, “Emergence and characterization of sex differences in spatial ability: A meta-analysis,” *Child development*, pp. 1479–1498, 1985.

[38] N. S. Newcombe and T. F. Shipley, “Thinking about spatial thinking: New typology, new assessments,” in *Studying visual and spatial reasoning for design creativity*, Springer, 2015, pp. 179–192.

[39] E. Deitrick, R. B. Shapiro, M. P. Ahrens, R. Fiebrink, P. D. Lehrman, and S. Farooq, “Using distributed cognition theory to analyze collaborative computer science learning,” in *Proceedings of the eleventh annual International Conference on International Computing Education Research*, 2015, pp. 51–60.

[40] J. S. Brown, A. Collins, and P. Duguid, “Situated cognition and the culture of learning,” *Educational researcher*, vol. 18, no. 1, pp. 32–42, 1989.

[41] E. Hutchins, “Distributed cognition,” *International Encyclopedia of the Social and Behavioral Sciences. Elsevier Science*, vol. 138, 2000.

[42] J. Hollan, E. Hutchins, and D. Kirsh, “Distributed cognition: Toward a new foundation for human-computer interaction research,” *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, no. 2, pp. 174–196, 2000.

[43] J. G. Greeno, A. M. Collins, L. B. Resnick, *et al.*, “Cognition and learning,” *Handbook of educational psychology*, vol. 77, pp. 15–46, 1996.

[44] M. S. Donovan, J. D. Bransford, and J. W. Pellegrino, “How people learn,” *Retrieved March*, vol. 8, p. 2006, 1999.

[45] L. S. Vygotsky, *Mind in society: The development of higher psychological processes*. Harvard university press, 1980.

[46] K. Charmaz and L. L. Belgrave, "Grounded theory," *The Blackwell encyclopedia of sociology*, 2007.

[47] K. Charmaz, *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[48] M. Muller, "Curiosity, creativity, and surprise as analytic tools: Grounded theory method," in *Ways of Knowing in HCI*, Springer, 2014, pp. 25–48.

[49] E. W. Dijkstra, *Computing science: Achievements and challenges*.

[50] R. McCauley, S. Grissom, S. Fitzgerald, and L. Murphy, "Teaching and learning recursive programming: A review of the research literature," *Computer Science Education*, vol. 25, no. 1, pp. 37–66, 2015.

[51] A. Settle, "What's motivation got to do with it? a survey of recursion in the computing education literature," 2014.

[52] C.-C. Wu, "Conceptual models and individual cognitive learning styles in teaching recursion to novices," 1993.

[53] H. Kahney, "What do novice programmers know about recursion," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM, 1983, pp. 235–239.

[54] P. Pirolli, "A cognitive model and computer tutor for programming recursion," *Human-Computer Interaction*, vol. 2, no. 4, pp. 319–355, 1986.

[55] J. Chao, D. F. Feldon, J. P. Cohoon, and J. Chao, "Dynamic Mental Model Construction : A Knowledge in Pieces-Based Explanation for Computing Students ' Erratic Performance on Recursion Dynamic Mental Model Construction : A Knowledge in Pieces-Based Explanation for Computing Students ' Erratic Performance on Recursion," *Journal of the Learning Sciences*, vol. 27, no. 3, pp. 431–473, 2018.

[56] S. Fincher, J. Jeuring, C. S. Miller, P. Donaldson, B. du Boulay, M. Hauswirth, A. Hellas, F. Hermans, C. Lewis, A. Mühling, *et al.*, "Notional machines in computing education: The education of attention," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 2020, pp. 21–50.

[57] E. W. Dijkstra, "Recursive programming," *Numerische Mathematik*, vol. 2, no. 1, pp. 312–318, 1960.

[58] T. R. Colburn and G. M. Shute, "Metaphor in computer science," *Journal of Applied Logic*, vol. 6, no. 4, pp. 526–533, 2008.

[59] I. Milne and G. Rowe, "Difficulties in learning and teaching programming—views of students and tutors," *Education and Information technologies*, vol. 7, no. 1, pp. 55–66, 2002.

[60] C. E. George, "Experiences with novices: The importance of graphical representations in supporting mental mode.," in *PPIG*, Citeseer, 2000, p. 3.

[61] T. Götschi, I. Sanders, and V. Galpin, *Mental models of recursion*, 1. ACM, 2003, vol. 35.

[62] I. Sanders, V. Galpin, and T. Götschi, "Mental models of recursion revisited," in *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, 2006, pp. 138–142.

[63] B. C. Bettin, "The stained glass of knowledge: On understanding novice mental models of computing," 2020.

[64] J. Sorva, "Notional machines and introductory programming education," *ACM Transactions on Computing Education*, vol. 13, no. 2, pp. 1–31, 2013.

[65] D. DeLiema and F. Steen, "Thinking with the body: Conceptual integration through gesture in multiviewpoint model construction," *Language and the Creative Mind. Borkent, Michael, Barbara Dancygier, and Jennifer Hinnell, editors. Stanford, CA: CSLI Publications*, 2013.

[66] D. Desutter, "Teaching students to think spatially through embodied actions : Design principles for learning environments in science , technology , engineering , and mathematics," 2017.

[67] G. Lakoff and M. Johnson, "Conceptual metaphor in everyday language," *The journal of Philosophy*, vol. 77, no. 8, pp. 453–486, 1980.

[68] L. W. Barsalou, "Grounded cognition," *Annu. Rev. Psychol.*, vol. 59, pp. 617–645, 2008.

[69] J. J. Gibson, *The ecological approach to visual perception: classic edition*. Psychology Press, 2014.

[70] A. Almjally, K. Howland, and J. Good, "Investigating children's spontaneous gestures when programming using tuis and guis," in *Proceedings of the Interaction Design and Children Conference*, 2020, pp. 36–48.

[71] M. W. Alibali, M. J. Nathan, M. S. Wolfgram, R. B. Church, S. A. Jacobs, C. Johnson Martinez, and E. J. Knuth, "How teachers link ideas in mathematics instruction using speech and gesture: A corpus analysis," *Cognition and instruction*, vol. 32, no. 1, pp. 65–100, 2014.

[72] M. W. Alibali, "Gesture in spatial cognition: Expressing, communicating, and thinking about spatial information," *Spatial cognition and computation*, vol. 5, no. 4, pp. 307–331, 2005.

[73] F. S. Azevedo and M. J. Mann, "Seeing in the dark: Embodied cognition in amateur astronomy practice," *Journal of the Learning Sciences*, vol. 27, no. 1, pp. 89–136, 2018.

[74] T. Kopcha and C. Ocak, "Embodiment of computational thinking during collaborative robotics activity," 2019.

[75] K. Ibrahim-Didi, M. W. Hackling, J. Ramseger, and B. Sherriff, "Embodied strategies in the teaching and learning of science," in *Quality teaching in primary science education*, Springer, 2017, pp. 181–221.

[76] S. W. Gilbert, "An evaluation," vol. 26, no. 4, pp. 315–327, 1989.

[77] G. Lakoff and M. Johnson, "M.(1980). metaphors we live by," *ChicagoLondon: University of ChicagoPress*, 1985.

[78] T. G. Amin, "Conceptual metaphor meets conceptual change," *Human Development*, vol. 52, no. 3, pp. 165–197, 2009.

[79] J. Wilbers and R. Duit, "On the micro-structure of analogical reasoning: The case of understanding chaotic systems," in *Research in science education-past, present, and future*, Springer, 2002, pp. 205–210.

[80] A. Manches, P. E. McKenna, G. Rajendran, and J. Robertson, "Identifying embodied metaphors for computing education," *Computers in Human Behavior*, vol. 105, p. 105 859, 2020.

[81] D. T. Brookes and E. Etkina, "Using conceptual metaphor and functional grammar to explore how language used in physics affects student learning," *Physical Review Special Topics - Physics Education Research*, vol. 3, no. 1, 2007.

[82] F. Jeppsson, J. Haglund, T. G. Amin, and H. Strömdahl, “Exploring the Use of Conceptual Metaphors in Solving Problems on Entropy Exploring the Use of Conceptual Metaphors in Solving Problems on Entropy,” vol. 8406, 2013.

[83] D. Gentner, “Structure-mapping: A theoretical framework for analogy,” *Cognitive science*, vol. 7, no. 2, pp. 155–170, 1983.

[84] R. Duit and M. Education, “The role of analogies and metaphors in learning science,” no. November 1991, 2015.

[85] D. Gowin, “Metaphors and conceptual change: Once more with feeling,” *Proceedings of the International Seminar Misconceptions in Science and Mathematics*, 1983.

[86] B. Dreyfus, A. Gupta, and E. Redish, “Applying conceptual blending to model coordinated use of multiple ontological metaphors,” *Conceptual metaphor and embodied cognition in science learning*, pp. 80–106, 2018.

[87] E. Ackermann, *Perspective-taking and object construction: Two keys to ...*

[88] E. Ochs, P. Gonzales, and S. Jacoby, “
when i come down im in the domain state”: Grammar and graphic representation in the interpretive activity of physicists,” *Interaction and Grammar*, pp. 328–369, 1996.

[89] P. Aubusson, S. Fogwill, R. Barr, and L. Perkovic, *What happens when students do simulation-role-play in science?*

[90] S. Goldin-Meadow, S. C. Levine, E. Zinchenko, T. K. Yip, N. Hemani, and L. Factor, “Doing gesture promotes learning a mental transformation task better than seeing gesture,” *Developmental science*, vol. 15, no. 6, pp. 876–884, 2012.

[91] S. Goldin-Meadow, “How gesture promotes learning throughout childhood,” *Child development perspectives*, vol. 3, no. 2, pp. 106–111, 2009.

[92] W.-M. Roth, “Gestures: Their role in teaching and learning,” *Review of educational research*, vol. 71, no. 3, pp. 365–392, 2001.

[93] A. Kendon, “Did gesture have the happiness to escape the curse at the confusion of babel,” *Nonverbal behavior: Perspectives, applications, intercultural insights*, pp. 75–114, 1984.

[94] ——, “Gesticulation and speech: Two aspects of the process of utterance,” *The relationship of verbal and nonverbal communication*, vol. 25, no. 1980, pp. 207–227, 1980.

[95] M. Novack and S. Goldin-Meadow, “Learning from gesture: How our hands change our minds,” *Educational psychology review*, vol. 27, no. 3, pp. 405–412, 2015.

[96] S. C. Broaders, S. W. Cook, Z. Mitchell, and S. Goldin-Meadow, “Making children gesture brings out implicit knowledge and leads to learning.,” *Journal of Experimental Psychology: General*, vol. 136, no. 4, p. 539, 2007.

[97] R. B. Church, S. Ayman-Nolley, and S. Mahootian, “The role of gesture in bilingual education: Does gesture enhance learning?” *International Journal of Bilingual Education and Bilingualism*, vol. 7, no. 4, pp. 303–319, 2004.

[98] M. Perry, D. Berch, and J. Singleton, “Constructing shared understanding: The role of nonverbal input in learning contexts,” *J. Contemp. Legal Issues*, vol. 6, p. 213, 1995.

[99] L. Valenzeno, M. W. Alibali, and R. Klatzky, “Teachers’ gestures facilitate students’ learning: A lesson in symmetry,” *Contemporary Educational Psychology*, vol. 28, no. 2, pp. 187–204, 2003.

[100] R. M. Ping and S. Goldin-Meadow, “Hands in the air: Using ungrounded iconic gestures to teach children conservation of quantity.,” *Developmental psychology*, vol. 44, no. 5, p. 1277, 2008.

[101] M. W. Alibali and M. J. Nathan, “Embodiment in mathematics teaching and learning: Evidence from learners’ and teachers’ gestures,” *Journal of the learning sciences*, vol. 21, no. 2, pp. 247–286, 2012.

[102] R. M. Ping, S. Goldin-Meadow, and S. L. Beilock, “Understanding gesture: Is the listener’s motor system involved?” *Journal of Experimental Psychology: General*, vol. 143, no. 1, p. 195, 2014.

[103] N. M. McNeil and D. H. Uttal, “Rethinking the use of concrete materials in learning: Perspectives from development and education,” *Child development perspectives*, vol. 3, no. 3, pp. 137–139, 2009.

[104] M. A. Novack, E. L. Congdon, N. Hemani-Lopez, and S. Goldin-Meadow, “From action to abstraction: Using the hands to learn math,” *Psychological Science*, vol. 25, no. 4, pp. 903–910, 2014.

[105] D. Efron and S. van Veen, *Gesture, race and culture*. 1972.

[106] P. Ekman and W. V. Friesen, "The repertoire of nonverbal behavior: Categories, origins, usage, and coding," *semiotica*, vol. 1, no. 1, pp. 49–98, 1969.

[107] B. Rimé and L. Schiaratura, "Gesture and speech.," 1991.

[108] D. McNeill, *Hand and mind: What gestures reveal about thought*. University of Chicago press, 1992.

[109] D. Kirsh, *Embodied cognition and the magical future of interaction ...*

[110] S. Wright, "Drawing and embodiment,"

[111] B. L. Sherin, "How students understand physics equations," *Cognition and instruction*, vol. 19, no. 4, pp. 479–541, 2001.

[112] J. Sorva, "Visual program simulation in introductory programming education,"

[113] I. Arawjo, "To write code: The cultural fabrication of programming notation and practice," *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020.

[114] B. Du Boulay, "Some difficulties of learning to program," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 57–73, 1986.

[115] T. Colburn and Æ. G. Shute, "Abstraction in Computer Science," no. June, pp. 169–184, 2007.

[116] E. Euler, E. Rådahl, and B. Gregorcic, "Embodiment in physics learning: A social-semiotic look," *Physical Review Physics Education Research*, vol. 15, no. 1, p. 010 134, 2019.

[117] K. Desportes, M. Spells, and B. Disalvo, "The movelab," *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016.

[118] G. Aranda and J. P. Ferguson, "Unplugged programming: The future of teaching computational thinking?" *Pedagogika*, vol. 68, no. 3, 2018.

[119] U. Leron and R. Zazkis, "Computational recursion and mathematical induction," *For the Learning of Mathematics*, vol. 6, no. 2, pp. 25–28, 1986.

[120] S. Papert, "Mindstorms: Children, computers, and powerful ideas," *The English Journal*, vol. 71, no. 8, p. 60, 1982.

[121] A. G. Soosai Raj, K. Ketsuriyonk, J. M. Patel, and R. Halverson, "Does native language play a role in learning a programming language?" In *Proceedings of the 49th ACM technical symposium on computer science education*, 2018, pp. 417–422.

[122] S. Vogel, "Translanguaging about, with, and through code and computing: Emergent bi/multilingual middle schoolers forging computational literacies," 2020.

[123] D. Pérez-Marın, R. Hijón-Neira, A. Bacelo, and C. Pizarro, "Can computational thinking be improved by using a methodology based on metaphors and scratch to teach computer programming to children?" *Computers in Human Behavior*, vol. 105, p. 105 849, 2020.

[124] F. Halasz and T. P. Moran, "Analogy considered harmful," in *Proceedings of the 1982 conference on Human factors in computing systems*, 1982, pp. 383–386.

[125] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–24, 2017.

[126] S. Watt, "Syntonicity and the psychology of programming.," in *PPIG*, 1998, p. 10.

[127] M. W. Alibali, S. Kita, and A. J. Young, "Gesture and the process of speech production: We think, therefore we gesture," *Language and cognitive processes*, vol. 15, no. 6, pp. 593–613, 2000.

[128] P. Seedhouse, "Conversation analysis and language learning," *Language teaching*, vol. 38, no. 4, pp. 165–187, 2005.

[129] M.-S. Seo and I. Koshik, "A conversation analytic study of gestures that engender repair in esl conversational tutoring," *Journal of Pragmatics*, vol. 42, no. 8, pp. 2219–2239, 2010.

[130] E. M. Crowder, "Gestures at work in sense-making science talk," *The Journal of the Learning Sciences*, vol. 5, no. 3, pp. 173–208, 1996.

[131] B. R. Shapiro and R. Hall, "Making engagement visible: The use of mondrian transcripts in a museum," in, Philadelphia, PA: International Society of the Learning Sciences., 2017.

[132] M. J. Nathan, “An embodied cognition perspective on symbols, gesture, and grounding instruction,” *Symbols and embodiment: Debates on meaning and cognition*, vol. 18, pp. 375–396, 2008.

[133] F. Vafaei, “Taxonomy of gestures in human computer interaction,” 2013.

[134] M. Chu and S. Kita, “The nature of gestures’ beneficial role in spatial problem solving.,” *Journal of Experimental Psychology: General*, vol. 140, no. 1, p. 102, 2011.

[135] R. Núñez, “Do real numbers really move? language, thought, and gesture: The embodied cognitive foundations of mathematics,” in *18 unconventional essays on the nature of mathematics*, Springer, 2006, pp. 160–181.

[136] T. R. Colburn and G. M. Shute, “Metaphor in computer science,” *Journal of Applied Logic*, vol. 6, no. 4, pp. 526–533, 2008.

[137] T. Sirkiä and J. Sorva, “Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises,” in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, ACM, 2012, pp. 19–28.

[138] A. Ebrahimi, “Novice programmer errors: Language constructs and plan composition,” *International Journal of Human-Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994.

[139] U. Wilensky, “Abstract meditations on the concrete and concrete implications for mathematics education,” in *Constructionism*, I. Harel and S. Papert, Eds., Norwood, NJ: Ablex, 1991, pp. 193–203.

[140] T. Colburn and G. Shute, “Type and metaphor for computer programmers,” *Techné: Research in Philosophy and Technology*, 2017.

[141] K. Devlin, “Require,” *Communications of the ACM*, vol. 46, no. 9, p. 37, 2003.

[142] J. Sorva, “Notional Machines and Introductory Programming Education,” vol. 13, no. 2, 2013.

[143] S. Fincher, J. Jeuring, C. S. Miller, B. Boulay, C. Lewis, and A. Petersen, “Notional Machines in computing education : The education of attention What is a Notional Machine,”

[144] M. A. Holliday and D. Luginbuhl, "Cs1 assessment using memory diagrams," in *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 2004, pp. 200–204.

[145] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, *et al.*, "A multi-national study of reading and tracing skills in novice programmers," *ACM SIGCSE Bulletin*, vol. 36, no. 4, pp. 119–150, 2004.

[146] P. Burton, "Kinds of language, kinds of learning," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 53–61, 1998.

[147] C. Goodwin, "The co-operative, transformative organization of human action and knowledge," *Proceedings of the 14th ACM international conference on Multimodal interaction - ICMI 12*, 2012.

[148] S. Scopelitis and R. Stevens, "Made by hand: Gestural practices for the building of complex concepts in face-to-face, one-on-one learning arrangements," 2010.

[149] R. Tytler, V. Prain, and P. Hubber, "Deakin Research Online," pp. 83–107, 2013.

[150] R. Tytler, K. Murcia, C.-T. Hsiung, and J. Ramseger, "Reasoning through representations," in *Quality teaching in primary science education*, Springer, 2017, pp. 149–179.

[151] M. Hackling, K. Murcia, and K. Ibrahim-Didi, "Teacher orchestration of multimodal resources to support the construction of an explanation in a year 4 astronomy topic," *Teaching Science*, pp. 7–15, 2013.

[152] S. Vossoughi, A. Jackson, S. Chen, W. Roldan, and M. Escudé, "Embodied pathways and ethical trails: Studying learning in and through relational histories," *Journal of the Learning Sciences*, vol. 29, no. 2, pp. 183–223, 2020.

[153] R. Barwell, "Formal and informal mathematical discourses: Bakhtin and vygotsky, dialogue and dialectic," *Educational Studies in Mathematics*, vol. 92, no. 3, pp. 331–345, 2015.

[154] M. L. Franke, E. Kazemi, and D. Battey, "Mathematics teaching and classroom practice," *Second lfaJU!/wak aflI.eseurch on Mathematics Tcachillg ali*, pp. 225–256, 2007.

[155] C. S. Peirce, R. S. Robin, *et al.*, "Charles s. peirce papers," 1963.

[156] C. M. Lewis, "Exploring variation in students' correct traces of linear recursion," in *Proceedings of the tenth annual conference on International computing education research*, ACM, 2014, pp. 67–74.

[157] J. G. Trafton, S. B. Trickett, and F. E. Mintz, "Connecting internal and external representations: Spatial transformations of scientific visualizations," *Foundations of Science*, vol. 10, no. 1, pp. 89–106, 2005.

[158] J. G. Trafton, S. B. Trickett, C. A. Stitzlein, L. Saner, C. D. Schunn, and S. S. Kirschenbaum, "The relationship between spatial transformations and iconic gestures," *Spatial cognition and computation*, vol. 6, no. 1, pp. 1–29, 2006.

[159] M. Stieff and S. Raje, "Expert algorithmic and imagistic problem solving strategies in advanced chemistry," *Spatial Cognition & Computation*, vol. 10, no. 1, pp. 53–81, 2010.

[160] P. Group, "Mip: A method for identifying metaphorically used words in discourse," *Metaphor and Symbol*, vol. 22, no. 1, pp. 1–39, 2007.

[161] B. Haberman and H. Averbuch, "The case of base cases: Why are they so difficult to recognize? student difficulties with recursion," in *Proceedings of the 7th annual conference on innovation and technology in computer science education*, 2002, pp. 84–88.

[162] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of learning in novice programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 37–55, 1986.

[163] J. P. Smith III, A. A. DiSessa, and J. Roschelle, "Misconceptions reconceived: A constructivist analysis of knowledge in transition," *The journal of the learning sciences*, vol. 3, no. 2, pp. 115–163, 1994.

[164] J. A. Roberts, "Connecting visitors to data: Exploring tools for mediating learning talk at an interactive museum exhibit," Ph.D. dissertation, University of Illinois at Chicago, 2016.

[165] J. Roberts and L. Lyons, "Examining spontaneous perspective taking and fluid self-to-data relationships in informal open-ended data exploration," *Journal of the Learning Sciences*, vol. 29, no. 1, pp. 32–56, 2020.

[166] A. Sfard, "On two metaphors for learning and the dangers of choosing just one," *Educational researcher*, vol. 27, no. 2, pp. 4–13, 1998.

[167] C. Paechter, "Metaphors of space in educational theory and practice," *Pedagogy, culture and society*, vol. 12, no. 3, pp. 449–466, 2004.

[168] D. Ancona, "Framing and acting in the unknown," *S. Snook, N. Nohria, & R. Khurana, The Handbook for Teaching Leadership*, pp. 3–19, 2012.

[169] K. E. Weick, K. M. Sutcliffe, and D. Obstfeld, "Organizing and the process of sensemaking," *Organization science*, vol. 16, no. 4, pp. 409–421, 2005.

[170] G. Fauconnier and M. Turner, "Conceptual integration networks," *Cognitive science*, vol. 22, no. 2, pp. 133–187, 1998.

[171] G. Fauconnier and G. Lakoff, "On metaphor and blending," *Cognitive Semiotics*, vol. 5, no. 1-2, pp. 393–399, 2009.

[172] N. S. Podolefsky and N. D. Finkelstein, "Analogical scaffolding and the learning of abstract ideas in physics : An example from electromagnetic waves," pp. 1–12, 2007.

[173] M. C. Wittmann, B. Hall, and O. Me, "Using conceptual blending to describe emergent meaning in wave propagation," vol. 1, pp. 659–666, 2010.

[174] N. Enyedy, M. Hall, L. Angeles, J. A. Danish, N. R. Ave, D. Deliema, M. Hall, and L. Angeles, "Constructing and Deconstructing Materially-Anchored Conceptual Blends in an Augmented Reality Collaborative Learning Environment," vol. 1, pp. 192–199, 2013.

[175] D. Silvis, V. R. Lee, J. Clarke-midura, J. Shumway, and J. Kozlowski, "Blending Everyday Movement and Representational Infrastructure : An Interaction Analysis of Kindergarteners Coding Robot Routes," pp. 98–105, 2020.

[176] R. G. Hausmann, M. T. Chi, and M. Roy, "Learning from collaborative problem solving: An analysis of three hypothesized mechanisms," in *Proceedings of the Annual Meeting of the Cognitive Science Society*, vol. 26, 2004.

[177] M. C. Parker, M. Guzdial, and S. Engleman, "Replication, validation, and use of a language independent cs1 knowledge assessment," in *Proceedings of the 2016 ACM conference on international computing education research*, ACM, 2016, pp. 93–101.

[178] C. M. Lewis, "Applications of out-of-domain knowledge in students' reasoning about computer program state," Ph.D. dissertation, UC Berkeley, 2012.

[179] H. Jeong and M. T. Chi, "Construction of shared knowledge during collaborative learning," 1997.

[180] E. Etkina and A. Warren, "Physics Instruction," no. December 2013, 2006.