Phenomenological Primitives in Introductory Computer Science Students' Understanding of Recursion

----

A Dissertation

Presented to

The Faculty of the Curry School of Education

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

----

by

Jie Chao

August 2012

ABSTRACT

Advisor: David F. Feldon, Ph.D.

Recursion is a difficult concept to learn in introductory computer science courses. Students frequently construct maladaptive mental models of recursion that interfere with their performance and subsequent skill development. Common explanations assume that these mental models are not decomposable mental structures. However, such an assumption fails to account for the inconsistent manifestation of these mental models across similar tasks.

This study applies the *knowledge-in-pieces* perspective (diSessa, 1993) to explain students' inconsistent performance on evaluation of recursive function. According to this perspective, *phenomenological primitives* (*p-prims*), experientially acquired tacit elemental knowledge structures, play dominant roles in naïve knowledge systems. Various task features may differentially constrain their influence, which renders them productive in some instances and problematic in others. This subtle mechanism gives rise to the inconsistent performance across tasks that target the same concept.

Reanalysis of data from previous studies suggests a potential p-prim that plausibly accounts for students' inconsistent performance within and across similar tasks. This p-prim reflects intuitive understandings of agentive causality (i.e. *agent* takes an *action* on a *patient* to generate certain *effect*) that commonly account for misunderstandings in physics concepts (diSessa, 1993). To evaluate this general hypothesis of a computer-as-agent p-prim, participants completed four tasks representing varying levels of constraint on their reasoning and participated in clinical interviews to report and explain their thought processes. It was expected that more participants would demonstrate the normative mental models of recursion in the high-constraint tasks than in the low-constraint tasks, because the computer-as-agent p-prim would be more likely to interfere with appropriate analysis under lower constraint. Further, participants' interpretations of the recursive functions were expected to demonstrate characteristics associated with p-prim-generated interpretations.

Results largely support the hypothesized p-prim. Participants' inconsistent performances were successfully explained by various modes of coordination between the computer-as-agent p-prim and relevant programming schemas. This finding advanced our understanding of students' difficulties in learning recursive programming and pointed to ways to improve instructional practices.

Department of Curriculum, Instruction, and Special Education
Curry School of Education
University of Virginia
Charlottesville, Virginia

APPROVAL OF THE DISSERTATION

This dissertation, *Phenomenological Primitives in Introductory Computer Science Students' Understanding of Recursion*, has been approved by the Graduate Faculty of the Curry School of Education in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

David F. Feldon, Ph.D., Advisor and Chair

Daniel B. Berch, Ph.D., Committee Member

James Cohoon, Ph.D., Committee Member

Daniel Willingham, Ph.D., Committee Member

6/18/12 Date

DEDICATION

I dedicate this dissertation to my wonderful parents, Yichuan Chao and Xiaodan Zhang. You have been so supportive of every educational decision that I’ve made. I appreciate that you always have faith in me. I also dedicate this dissertation to my loving and patient husband Xin Liao. Thank you for being such a wonderful part of my doctoral journey.

ACKNOWLEDGEMENTS

My deepest gratitude is to my advisor, Dr. David Feldon, who trusted in my intellectual capacity, gave me the freedom to explore on my own, and offered invaluable support and guidance. He has been a great mentor and great friend who has always understood my struggles and consistently inspired me. I hope that one day I would become as good an advisor to my students as David has been to me.

My thanks also go to the members of my dissertation committee. Dr. Jim Cohoon had been so supportive throughout my data collection process. He and his teaching assistant Luther Tychonievich kindly helped me recruit participants from his class and revised my test items with great patience. I am deeply grateful to Jim for the precious opportunity and generous help. Dr. Dan Berch and Dr. Dan Willingham had offered insightful feedbacks on my research and encouraged me to think more broadly and deeply. I appreciate that they hold me to a high research standard.

I am also indebted to Dr. Joanne Cohoon for offering me research assistantship in the first three years of my doctoral study. I survived financially with her help, and I learned many valuable research skills from her. Most importantly, her positive attitude, her passion, and her strength inspired me a great deal.

Special thanks also to all my colleagues and graduate friends for their support during my graduate study: Dr. Melissa Hurst, Dr. Michelle Maher, Dr. Briana Timmerman, Denise Strickland, Deepalakshmi Muniraj, Christopher Rates, Dr. Billie Eilam, Dr. Jennifer Chiu, Peter Malcolm, Edward Pan, Dr. Zhen Wu, Dr. Kate Cooke, and Feng Rao.

Last but not least, I would like to thank my parents and my husband for their understanding, patience, and infinite love during the past few years. None of this would have been possible without their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The United States is actively engaged in a large-scale movement promoting educational excellence in science, technology, engineering and mathematics (STEM) disciplines. The endeavor, currently motivated by a potential human resource shortage, is attributed in large part to an educational deficit in related disciplines (Augustine et al., 2005). The situation in the field of computer science (CS) is particularly urgent (Computing Research Association, 2011): despite a slow increase in the number of computing graduates over the past two years, institutions of higher education still cannot meet the fast growing demand for qualified graduates in the computing industry (Bureau of Labor Statistics, 2009).

As the main entry point to computing careers, CS introductory courses suffer from high attrition and failure rates (e.g. Beaubouef, 2005; Chinn, Martin, & Spencer, 2007; Guzdial & Soloway, 2002; Howles, 2007, 2009; McKinney & Denton, 2004). Also, at the conclusion of their introductory programming courses, empirical assessments indicate that many students are unable to perform standard programming tasks (McCracken et al., 2001). A nationwide survey of CS faculty members reports their perception that students have difficulties in understanding basic programming concepts such as parameters, functions, arrays, recursion, and object-oriented constructs. Further, the majority indicate that those students who do have an adequate understanding stumble when applying these concepts during problem solving (Dale, 2006).

Recursion is frequently cited as one of the most difficult topics for beginning CS students (e.g., Dale, 2006; Goldman et al., 2008). Most students have encountered neither the concept nor its vocabulary previously (e.g., Kahney, 1983; Levy, 2001). When presented with recursive phenomena such as Koch Snowflakes, students tend to interpret them only as repetitive, cyclic, sequential, or multi-dimensional, which do not convey the essence of the concept as a self-referencing function (Levy, 2001). Further, students' lack of understanding about how recursive functions are executed as programming code often persists through substantial training (Götschi, Sanders, & Galpin, 2003; Sanders, Galpin, & Götschi, 2006). Even when students are able to appropriately apply recursive syntax, they continue to have difficulty analyzing problems that require a recursive approach (Mirolo, 2010). If not explicitly suggested, many students will not employ recursion to solve inherently recursive problems (Ginat, 2004; Vilner, Zur, & Gal-Ezer, 2008).

Given these difficulties, many CS faculty members opt not to teach recursion in introductory courses at all (Dale, 2006). However, recursion's fundamental status in computing warrants an early placement in the curriculum (Schwill, 1994). Experts in introductory computing subjects consistently rank the importance of the concept highly (Goldman et al., 2008). The ACM/IEEE Joint Task Force on Computing Curricula consistently recommends coverage of recursion in introductory courses (The Joint Task Force on Computing Curricula, 2001; Tucker, 1991). Inclusion of recursion in many widely adopted introductory CS textbooks reflects implicit agreement from their authors about its importance (Tew & Guzdial, 2010). Although educational trends in CS have brought about six different tracks of introductory courses (i.e. imperative-first, objects-first, functional-first, breadth-first, algorithms-first, and hardware-first), recursion is one of only a few topics shared across all tracks (Shackelford et al., 2006). Thus, a thorough understanding of the factors underlying students' learning difficulties could inform instructional practices and substantially benefit the preparation of the future computing workforce.

Recursion

In the most general sense, recursion is a method to define something in terms of itself. The classic example frequently used to introduce recursion is the factorial function. In mathematics, the factorial of a positive integer n (i.e., n!) is the product of all positive integers less than or equal to n (given that 0! = 1). For example, 3! = 1*2*3 = 6. In the context of computer programming, a *recursive function*[1] can be written to generate the factorial value of any input n. From the problem solving perspective, recursion can be defined as a method that "breaks a problem down into smaller sub-problems of the same kind…the solutions to the sub-problems are then combined into a solution to the main problem" (Holyer, 1991, p.51). According to this definition, the smaller problem instance of the factorial function n! is (n-1)!. Thus, n! can be obtained by applying the operation "multiply n" to (n-1)!. In addition, the smallest instance of the problem, or the *base case*,

[1] Recursive function is called by different names in different programming environments such as recursive method in Java (Cohoon & Davidson, 2006), recursive function in LISP (Anderson, Pirolli, & Farrell, 1988), and recursive procedure in Scheme (Abelson, Sussman, & Sussman, 1996). Sometimes, it is also called recursive algorithm (Götschi, 2003; Götschi, Sanders, & Galpin, 2003). In order to simplify communication, the term "recursive function" is used instead of its variants in the review of literature.

for which the answer is readily available is 0!=1. Therefore, the recursive function can be expressed by the following pseudo code:

```
factorial (n) {
 If (n=0) result 1;                  *base case
 else result n*factorial (n-1);*factorial (n-1) is a smaller problem instance
}
```

The way that the function employs itself as part of the steps is known as the *recursive structure*. However, the above definition does not make evident how the computer executes the function. To interpret the factorial (n-1) within the factorial (n), the computer itself relies on the other definition of recursion: “a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones” (SOLO Programming Manual, as cited by Kahney, 1983, p.235). To illustrate this process, a representation of how the computer interprets and executes factorial (3) is shown in Figure 1.

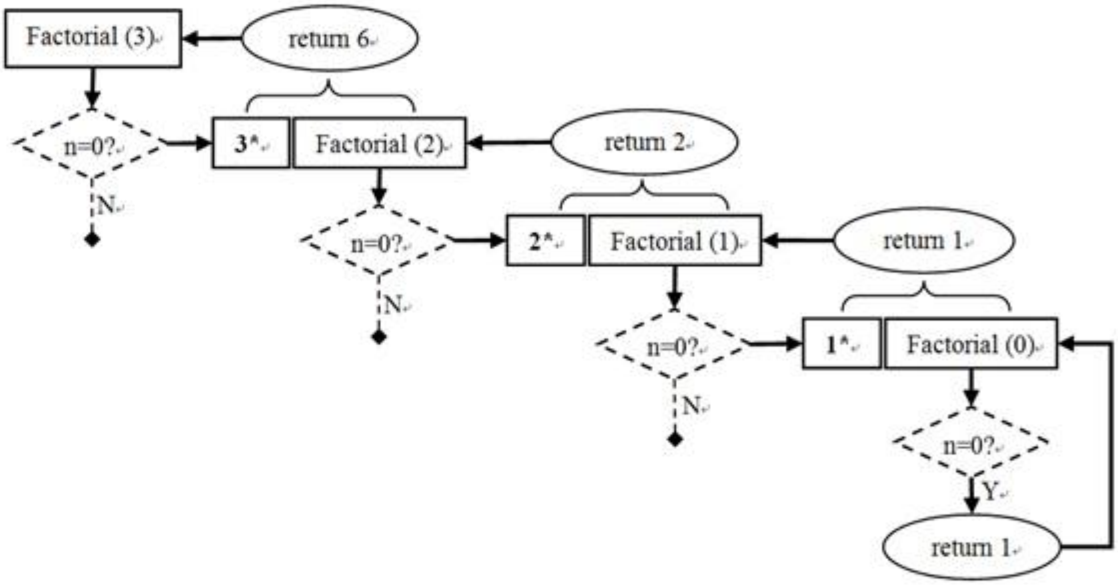


Figure 1. Simulation of how computer executes the factorial function with an input of 3.

In Figure 1, the *recursive process* (denoted by the arrows) starts with processing factorial (3). Firstly, the argument 3 is evaluated against the condition n=1? (denoted by the diamond frames). If n does not equal 1, then the second line of the function 3*factorial (2) is processed. The operation n* (denoted by the short rectangular frames), however, will be suspended because the *recursive call*, factorial (2) (denoted by the long rectangular frames) needs to be instantiated and processed in the same manner. When the processing reaches the stopping rule, factorial (0), for which the result (denoted by the oval frames) is already known, the result will be passed back to the previously suspended operations until the result of the original problem, factorial (3), is obtained (Abelson, Sussman, & Sussman, 1996). The process prior to the base case is called active flow of control, and the process after is called passive flow of control (George, 2000).

Noticeably, there are two sets of terms that define recursion: 1) *base case* and *smaller problem instance*, and 2) *stopping rule* and *recursive call*. These two definitions and associated terms are not simply different wordings. They reflect a fundamental distinction between the functional and the procedural programming paradigms (Haberman & Averbuch, 2002). Functional programming expresses the logic of a computation without describing its flow of control, and it treats computation as the evaluation of mathematical functions. Such a function only describes what should be accomplished. In contrast, procedural programming describes a computation with statements that define sequences of commands for the computer to perform. Because recursion originated within the functional programming paradigm, programmers may only attend to the formulation of the necessary recursive structure itself. However, understanding the recursive process is also important for verifying and debugging complex recursive functions (Segal, 1995).

There are many types of recursive functions, though only those relevant to the current study are introduced here. *Tail recursion* is a special case of recursion, in which the recursive call is immediately followed by a return. In other words, the value obtained from the called procedure must be returned by the calling procedure. For example, the factorial function can be written into a tail recursion as:

```
fact (n, accumulator) {
    if (n == 0) return accumulator;
    return fact (n - 1, n * accumulator);
  }

factorial (n) {
    return fact (n, 1);
}
```

In the auxiliary function fact (n, accumulator), the parameter “accumulator” is responsible for recording the changing result so that no suspended operation is needed. As such, tail recursive functions generate an iterative process, which can also be generated by an equivalent, non-recursive looping function.

In contrast to tail recursion, *embedded recursion* is complete and typical in the sense that there are operations both preceding and following the recursive call. The operations that precede the recursive call are called *head-block*, and those that follow the recursive call are called *tail-block* (Kurland & Pea, 1985; diCheva & Close, 1996). In order to enable fine-grained analysis in this study, head-block or *head* and tail-block or *tail* only refers to operations that are independent of the recursive instantiations. Operations contingent on the results of recursive calls are referred to as *prefix* or *suffix* depending on their locations relative to the recursive calls.

Statement of the Problem

Students frequently construct maladaptive mental models of recursion that interfere with their skill development and subsequent performance (e.g., Kahney, 1983; Scholtz &

Sanders, 2010). Common explanations assume that these maladaptive mental models are not decomposable mental structures developed from prior learning of related looping constructs (e.g., Turbak, Royden, Stephan, & Herbst, 1999) or incompatible programming paradigms (e.g., Ginat & Shifroni, 1999). These explanations, however, cannot account for several sets of empirical findings. First, individuals' mental models and associated misconceptions manifest inconsistently across problem contexts (Mirolo, 2010; Segal, 1995), so they cannot have a single, stable cause. Second, empirical evidence reflects significant benefits to learning of looping constructs prior to developing recursive programming skills (Kessler & Anderson, 1986; Wiedenbeck, 1988) and instances where programming paradigms appropriate for recursion also invoke misconceptions (Kahney, 1983; Segal, 1995). Third, cognitive development is continuous, which belies the definition of a misconception as a discrete but mistaken knowledge state (Smith, diSessa, & Roschelle, 1994).

In order to better explain students' difficulties in mastering recursive programming skills, this study applies the *Knowledge–in-Pieces* theory (diSessa, 1988, 1993). This theory accounts for the inconsistency, origination, and development of maladaptive knowledge in the domain of physics education. According to the knowledge-in-pieces theory, conceptual knowledge is best understood as a complex system that comprises parts whose interactions generate emergent global properties. The naïve knowledge system is characteristic of loosely connected knowledge structures, whereas in the expert knowledge system these knowledge structures are properly connected to achieve coordinated applications.

There are a variety of knowledge structures with varying degree of structural complexity and different origination. The most fundamental knowledge structures, *phenomenological primitives* (p-prims; diSessa, 1993), are composed of only a few parts and develop from experiences with common phenomena. For example, a p-prim called "force as a mover" forms from the experiential phenomenon that motion is caused by a push (diSessa, 1993, p.129). A p-prim is activated only when the configuration of the contextual features fits is designated circumstance. A slight change in the contextual configuration may activate a different p-prim. For instance, the "force as a mover" p-prim cannot respond to other circumstances like a moving object slowing down to a stop, which has its own responsible p-prim "dying away" (diSessa, 1993, p.133). As such, a large number of p-prims exist for numerous common physical phenomena. They enable people to explain and predict events in the world in a way that is intuitive and consistent with lived experience, even if the p-prims are not fully consistent with natural laws. With the "force as a mover" p-prim, a person seeing someone kicking a ball will be able to instantly predict the ball's subsequent motion due to operation of the p-prim. Thus, p-prims underpin our fluency in interacting with physical world.

An important characteristic of p-prims is that they play different roles in knowledge systems with varying levels of expertise. In a naïve knowledge system, their activation is largely independent, rarely spreading to other knowledge structures. This explains the observation that students often feel satisfied with their intuitive explanation of physical phenomena instead of relating to the scientific concepts for deeper explanation (e.g., diSessa, 1993). In contrast, a p-prim in an expert knowledge system does not operate as an independent explanatory construct, but as a trigger to spread activation to a cluster of knowledge structures, referred to as a *coordination class* (diSessa & Sherin, 1998).

Coordination classes correspond to scientific concepts (e.g., force, heat, etc.) if those concepts are properly understood. Thus, the process of conceptual change, which is central to expertise acquisition, is viewed as reorganization among p-prims and interconnections among relevant knowledge structures in response to environmental perturbations.

The knowledge-in-pieces theory appears to be a promising explanation of students' misconceptions of recursion. Given the hypothesis that a misconception is the product of a functioning p-prim, the knowledge-in-pieces theory would predict erratic performance across isomorphic problem contexts, because the surface differences between the contexts may activate different p-prims or change p-prims' role in activation networks. Further, the existence of relevant p-prims prior to formal instruction in recursion can explain misconceptions that occur regardless of the compatibility of previously learned looping constructs or programming paradigms. In addition, the conceptual change mechanism specified by the knowledge-in-pieces theory would address the issue of discontinuity entailed by viewing misconceptions as context-independent beliefs. A knowledge-in-pieces account of recursion learning would ultimately specify the p-prims underlying common misconceptions and describe the conceptual transformation processes in detail. Such a systematic and elaborated account would offer more specific recommendations for assessment and instructional intervention than those upon which current efforts are based (e.g., Scholtz & Sanders, 2010).

Purpose of the Study

The purpose of this study is to explain beginning CS students' misconceptions of recursion using the knowledge-in-pieces theory. To accomplish this, it is necessary to identify and characterize underlying p-prims. Thus, the study's research questions are:

1. Do beginning CS students demonstrate reliance on identifiable p-prims when trying to understand and apply recursion?
2. If they do, what are the structures, relevant circumstances, functions, and effects of these p-prims as they impact learning and performance?

Significance of the Study

This study offers an alternative explanation for students' misconceptions of recursion. As indicated in the statement of the problem, the existing alternative views are inadequate for explaining the inconsistency, origination, and development of misconceptions. The knowledge-in-pieces theory can account for these phenomena.

A knowledge-in-pieces-based explanation holds implications for the design of diagnostic assessment. Traditionally, the goal of diagnostic assessment is to detect the misconceptions held by students. However, misconceptions are limited targets for intervention due to their inconsistent manifestations across problems. The fine-grained and context-bound p-prims may offer more meaningful diagnosis because of their links to both the surface and deep structures of problems.

Further, findings from this study can provide a basis for a comprehensive model of recursion learning. On the basis of identified p-prims, future studies can more fully examine the evolution of the knowledge base that supports recursive programming. As the structures, functions, and activating contexts of p-prims are better understood, a more nuanced and targeted instructional approaches can foster the efficient and accurate configuration of p-prims into coordination classes to support high-level performance in CS. Specifically, these findings can inform three current debates in the CS education community.

The first debate pertains to whether recursive process, recursive structure, or both should be taught in introductory CS courses. The dominant view among CS educators emphasizes only a general understanding of the recursive process (e.g., Götschi et al., 2003; Sanders et al., 2006; diCheva & Close, 1996). However, some researchers seriously question the adequacy of emphasizing only this aspect. Ginat and Shifroni (1999) and Haberman (2004) demonstrate that understanding the recursive process is inadequate for students to generate recursive functions. They argue in favor of an emphasis on recursive structure to enhance students' abilities to solve recursive problems. Others advocate a purely declarative instructional approach by showing that one can use an inductive method to construct and verify recursive functions without considering how the functions are executed by the computer (Ford, 1984).

The second debate pertains to the optimal sequence for teaching programming concepts. Given the conflicting findings regarding the relationship between knowledge of looping construct and knowledge of recursion, there are divided opinions regarding which should be taught first. For functional programming languages such as Scheme, recursion is consistently introduced earlier because looping can only be expressed via tail recursion (e.g., Abelson, Sussman, & Sussman, 1996). With other programming languages, either approach is viable (e.g., Turbak et al., 1999). Most textbooks and instructors treat recursion as an advanced topic and place it after looping—usually toward the end of the introductory CS curriculum (e.g., Cohoon & Davidson, 2006; Savitch, 2008). Looping-first sequencing is supported by some findings that learning looping first benefits learning recursion but not vice versa (Kessler & Anderson, 1986; Wiedenbeck, 1988). However, other studies suggest that recursion should be taught first, because prior experience with looping interferes with learning of recursion (Levenick, 1990; Turbak et al., 1999).

These two debates can be reformulated by viewing these programming constructs – recursive process, recursive structure, looping construct and recursive construct – as coordination classes. According to knowledge-in-pieces theory, two coordination classes may be mutually exclusive or overlapping depending on whether they share the same knowledge structures (diSessa, 1993). This compositional relationship may influence these coordination classes' development trajectories. Thus, development of the coordination classes may be independent of each other, mutually facilitating, or mutually hindering at different phases of their development. These potential complexities may be responsible for the controversial views discussed above. Identifying relevant p-prims and their evolution into coordination classes over time can recommend one approach or a combination of approaches that target those p-prims most likely to interfere with subsequent knowledge acquisition.

The third debate concerns the selection of appropriate examples to introduce recursion. Typically, CS textbooks and instructors introduce recursion using mathematical functions such as factorials or Fibonacci numbers. However, these mathematical examples may neither be engaging to students nor intuitive to understand due to the mathematical knowledge required (Levenick, 1990; Turbak et al., 1999). Alternatively, introductory examples could be drawn from everyday events (Levenick, 1990; Turbak et al., 1999). However, these examples can be problematic due to students' intuitive and idiosyncratic interpretations (Conway & Kahney, 1987). In this respect, mathematical examples might be more appropriate because they are much less subject to individualized interpretations than everyday examples.

From the knowledge-in-pieces perspective, the choice may be situational. For students without adequate mathematical knowledge—especially the knowledge of mathematical induction—the non-mathematical example strategy is promising, because they are likely to activate p-prims that need to be incorporated to the coordination class of recursion. However, the existing p-prim activation patterns may hinder learning of recursive concepts. For students with prior knowledge of mathematical induction, the coordination class of recursion has already been developed in the domain of mathematics. Extending it to the domain of programming would likely not be as demanding. Thus, mathematical examples would be a reasonable choice in such situation.

Delimitations and Limitations

The objective of this study is to identify and characterize p-prims in introductory CS students' understanding of recursion, but the p-prim descriptions to be generated are preliminary. Because p-prims are hypothesized to play different roles as expertise develops, these working versions must be evaluated in knowledge systems at different levels of expertise. This present study, however, is only focused on naïve knowledge systems. To complete and refine the descriptions of p-prims, further studies will need to be conducted in more developed knowledge systems.

The methodology of this study limits its generalizability in terms of sample representativeness and performance representativeness. Participants are students at a top-ranked American university, so they are likely to differ in important ways from students with less demonstrated academic success. Also, these participants were recruited from a class that only enrolled students with no prior programming knowledge, thus they were likely to begin with lower programming competence than average introductory CS students who often have varied background in programming. Also, because participation is voluntary, the participants are likely to have different motivational characteristics than otherwise similar students who did not volunteer. The sampling frame limits the ability to generalize conclusions from the sample to the population, although the objective of this study is to conceptualize theoretical constructs rather than produce generalizable results.

CHAPTER 2

REVIEW OF LITERATURE

In order to meaningfully explore the problems posed in Chapter 1, it is first necessary to examine two sets of theoretical assumptions and their empirical supports. Specifically, the first set, the traditional cognitive information processing perspective, includes the assumptions that (a) learning is governed by a domain-general skill acquisition mechanism (e.g., Pirolli & Anderson, 1985) and (b) students have no prior knowledge in recursive function (Anderson, Farrell, & Sauers, 1984). The latter assumption permits the former to be applied to the studies on learning of recursive function without prior knowledge acting as a confounding variable. These two assumptions together render learning of recursion a mechanical process of acquiring information and rules from scratch. The associated empirical studies, however, evoke observations that refute the no-prior-knowledge assumption and problematize the skill acquisition assumption.

These observations give rise to the second set of assumptions aligning with the constructivist perspective, that (a) learning occurs when students construct new knowledge on the basis of their prior knowledge even when that knowledge is not specific to recursion (e.g., experience with the looping construct; Kahney & Eisenstadt, 1982; Kessler & Anderson, 1986) and (b) due to interference of the prior knowledge, students construct relatively stable mental models of recursion that consist of persistent, maladaptive misconceptions. Acknowledging the importance of students' prior and intermediate knowledge states, recent studies identify and characterize persistent types of mental models problematic for successful use of recursive functions. Despite consistent evidence of these mental model categories across studies, the literature has not yet recognized higher order patterns that occur as mental models manifest across various task situations.

The mental models described in the extant literature are examined through the lens of a knowledge-in-pieces conceptual framework (diSessa, 1993) to characterize common aspects of these models that may manifest differently across presented tasks. Data from these studies are reanalyzed for this purpose to identify potential p-prims that may play an active role in the understanding of recursion in computer science.

Information Processing Perspective: Skill Acquisition

The Adaptive Control of Thought-Rational (ACT-R) theory of learning (Anderson, 1993) serves as the foundation for a series of studies on learning of recursive function (Anderson, Pirolli, & Farrell, 1988; Pirolli, 1986, 1991; Pirolli & Anderson, 1985). ACT-R specifies basic cognitive units and operations that drive human cognition. In ACT-R, there are two distinct forms of knowledge: declarative and procedural. Declarative knowledge is the factual information stored in long-term memory, which is represented and manipulated in working memory in the form of a chunk—a primitive knowledge structure with only a few parts. Procedural knowledge, or the knowledge of when and how to perform a specific task, is embodied in the form of production rule. Production rules consist of initiating conditions coupled with actions that are initially acquired as declarative knowledge but evolve into a nondeclarative form through practice over time.

Knowledge acquisition is governed by two distinct mechanisms for declarative knowledge and procedural knowledge respectively. Declarative knowledge is acquired by encoding information from the environment or retrieving results of past mental processing. Procedural knowledge is acquired through analogical problem solving, in which learners formulate production rules on the basis of examples and apply them to solve problems.

Analogical Skill Acquisition Model of Learning Recursion

Based on the ACT-R theory, learning of recursive function is viewed as a skill acquisition process governed by the analogical problem solving mechanism (Pirolli & Anderson, 1985). Empirical studies show that participants initially rely heavily on examples and analogical problem solving strategies. If examples are provided, most students spend more than one third of their time looking at the examples during their first attempt at solving recursion problems (Pirolli & Anderson, 1985). Protocol analysis reflects students' attempts to solve problems by structurally mapping problem features onto those of the examples and then importing elements from the examples into the solutions. This analogical problem solving process is characterized in terms of production rule acquisition and compilation. In the first place, matching features between example and problem is achieved by a set of preexisting comparison productions. If an arbitrarily defined similarity criterion is met, a preexisting domain-general structure-mapping production is implemented to generate a sequence of actions that solve the problem (Singley & Anderson, 1989). Then, these problem-solving actions are compiled into production rules specific to the problem. With the acquired production rules, students depend less and less on examples as they progress through further practice. Also, if the similarity between the example and the problem is high, students are less likely to make errors on the parts of the problem not analogous to the example (Pirolli, 1991).

Flaws of the Analogical Skill Acquisition Model

However, research also shows that students often do not consider using given examples as analogies or misinterpret problems in ways that hinder analogical problem solving (Kahney & Eisenstadt, 1982). Further, the mapping strategies they employ often remain at the syntactical level and do not reflect the underlying functional concepts, resulting in poor understanding of both recursive function (Kessler & Anderson, 1986) and target problem (Kahney & Eisenstadt, 1982).

These results highlight problems with the assumption that students have no prior knowledge of recursion. Students likely do not have formal knowledge of recursion, but their intuitive assumptions about recursive function and recursive phenomena can influence learning processes and outcomes. They often interpret a recursive problem on the basis of their informal knowledge about the problem (Kahney & Eisenstadt, 1982; Levy, 2001), unintentionally fleshing out the problem statement with irrelevant elements drawn from their everyday knowledge about the problem cover stories (Keane, Kahney, & Brayshaw, 1989). The flaws of the analogical skill acquisition model, as evidenced in these findings, are accounted for in the studies on mental models of recursion reviewed in the following section.

Constructivist Perspective: Mental Models

Parallel to the inquiries on how students acquire recursive programming skills, many researchers investigate how students make errors in recursive programming. The data sources are those typically disregarded by skill acquisition research in the tradition of ACT-R. The observational settings are not as constrained as those in the skill acquisition experiments. Frequently, data are gathered from naturalistic settings such as course exams or classroom activities. This research explicitly or implicitly draws on the notion of a dynamic mental model to interpret observations.

Mental Model Theory

The theory of mental models provides a pragmatic explanation of human reasoning (Craik, 1943). In the most general sense, mental models are internal representations of certain domains. People construct, manipulate, and evaluate mental models to explain and predict events occurring in the world (Gentner & Stevens, 1983; Johnson-Laird, 1983). Although a pragmatic tool for reasoning and problem solving, mental models do not necessarily reflect the state of affairs in the world accurately and completely (Norman, 1983). Their flaws and imperfection account for human’s bias and errors in ordinary reasoning (Johnson-Laird, 1983). For instance, novices in formal logic do not apply rules of inference to make syllogistic inference. Instead, they build mental models of the given premises and then test their validity by directly manipulating the components of the models. Certain characteristics of premise sets render mental model more or less difficult to construct and manipulate, which lead to systematic judgment bias toward the simpler inference situations (Johnson-Laird, 1980).

Mental models are constantly evolving as people continuously interact with the world (Norman, 1983). This evolution may occur as model components are added, removed, or differentiated, or as relationships among model components are established, eliminated, or altered (Ifenthaler, Masduki, & Seel, 2011). This dynamic property of mental models makes it a useful construct to explain novice-expert knowledge gap (e.g., Larkin, 1983) and the mechanism of conceptual change (e.g., Chi, 2008; Clement & Steinberg, 2002). The nuanced properties of this representational system, however, are still subject to individual researcher's interpretation.

Mental Model Studies in Computer Science Education

CS education research largely applies mental model theory to understand novices' programming errors. Theoretically, a programming error is an instance in which a certain feature of a program deviates from that of a correct program. However, programs as functional entities cannot be simply compared line by line for surface differences. There are deep structural differences that need to be identified and explained (Spohrer & Soloway, 1986; Youngs, 1974). Therefore, researchers consider the programmers' mental structures and processes in order to better characterize and categorize programming errors.

In a widely accepted categorization scheme, programming errors are categorized into five types based on the "level of understanding needed to correct the errors" (Youngs, 1974, p.363): clerical, syntactic, semantic, logical and stylistic or discursive (du Boulay & O'Shea, 1981; Soloway & Ehrlich, 1984). Clerical errors refer to typographical errors and other unintended actions. Syntactic errors refer to mistakes in syntax usage. Semantic errors, now frequently referred to as runtime error, are syntactically correct but impossible or contradictory commands. Logical errors occur when a syntactically and semantically correct program does not fulfill the purpose of solving a target problem due to designing or planning failures. Lastly, stylistic or discourse errors are identified when a coding style violates the standard or convention of the programming community.

Among these, failures of recall or the limitations of working memory cause clerical and syntactic errors (Anderson, Farrell, & Sauers, 1984). Stylistic or discursive errors are attributed to lack of socialization into the programming community (Soloway & Ehrlich, 1984). Semantic errors and logical errors are considered the most difficult to remediate, because they are attributed to novices' poor understanding of programming constructs or problems and represented as maladaptive mental models (Youngs, 1974). As computer science educators are most concerned with these two fundamental errors, mental model studies are prevalent in the computing education literature (e.g., Ben-Ari, 1998; Ma, Ferguson, Roper, & Wood, 2011; Pea, Soloway, & Spohrer, 1987).

Mental Models of Recursion

Researchers conceptualize mental models of recursion differently in terms of the scope of target domains. In programming, the problem and the machine are two distinct domains, and programmers have mental models for each domain respectively (Isbell et al., 2010). Only a few researchers focus on the problem domain, or problem interpretation (e.g., da Rosa, 2007; Levy, 2001). More researchers concentrate on the machine domain, or the computational mechanism (e.g., Götschi et al., 2003; Kahney, 1983; Mirolo, 2010; Sanders et al., 2006; Scholtz & Sanders, 2010). However, other studies do not distinguish between the two domains or their associated mental models (Bhuiyan, Greer, & McCalla, 1994; Conway & Kahney, 1987; Ginat, 2004; Pirolli & Anderson, 1985; Vilner et al., 2008; diCheva & Close, 1996).

For instance, *loop* model, frequently found among novices, consists of a maladaptive mental model of the execution mechanism for recursive functions and a maladaptive mental model of a given problem (Bhuiyan et al., 1994; diCheva & Close, 1996). Students first misinterpret essentially recursive problems as governed by iterative rules thus mistakenly formulate iterative solutions. Meanwhile, they also misconceive recursion as a looping construct and suitable to express the iterative solutions. Unfortunately, such composite conceptualization hinders in-depth investigation of both, because it masks the essential differences between the two mental models (Young, 1983) and the mechanism that underlies their asynchronous development (Mirolo, 2010). In the current study, the mental models of recursion solely refer to mental models of the execution mechanism for recursive functions.

Researchers commonly use evaluation tasks to elicit participants' knowledge of recursive function. Typically, participants read given recursive functions, predict outputs for given inputs, and explain the processes in the form of trace (e.g., Götschi et al., 2003), through which they arrive at their answers. Some researchers also interview participants to explore or clarify their thought processes (Scholtz & Sanders, 2010). Participants' responses are analyzed to differentiate qualitatively different understandings of recursive function. Although proposed mental model categorizations slightly vary by coding schemes and specificity, overall they exhibit high levels of convergence described below.

Looping Model

Model profile. The *looping* model (Götschi et al., 2003; Kahney, 1983) refers to a view of recursive function as a looping construct. Specifically, this model involves a flawed execution mechanism. New parameter values are generated by operations before or within a recursive call. When the recursive call is reached, evaluation restarts from the first line of the function with the new parameter values, overriding the old ones. This procedure goes on until the base case is reached; then evaluation continues to other operations after the recursive call. A typical trace associated with the looping model, as shown in Figure 2, is characteristic of a sequence of function restarts with new parameter values at the active flow, stop of the flow at the base case, and no passive flow (Götschi, 2003).

```
factorial (3)
3*   factorial (2)
    2*    factorial (1)
        1*    factorial (0)
                     return: 1
```

Figure 2. A looping trace of the factorial function with an input of 3.

Associated misconceptions. In analysis of the looping model's structure, some researchers assert a general underlying misconception that a recursive call is a looping construct (Kahney, 1983; Kurland & Pea, 1985). Others propose one or more specific misconceptions that give rise to the looping model. For example, the looping model may be a product of repeatedly applying a specific misconception that the recursive call signals execution process to jump to the start of the procedure (Leonard, 1991). It may also be attributable to a misconception that the base case is a stopping condition for the execution process (Götschi, 2003; Segal, 1995) or a misconception that the recursive call is a static segment of program code (George, 2000b) or a single object (Götschi, 2003). Further, variants of the looping model are attributed to combinations of several misconceptions (diCheva & Close, 1996): 1) the command STOP—a language-specific command in Logo programming language—stops recursive procedure; 2) the command STOP stops overall execution; 3) the computer takes a variable as a non-changing construct throughout execution; 4) a variable is assigned with a new value by an operation; 5) a variable is assigned with new value at each newly initiated procedure; 6) the initial value of a variable is maintained for the tail-block.

Frequency and persistence. The looping model is common among novices. Kahney (1983), for example, reports that 53% of the 30 participants trained briefly in SOLO programming exhibited the looping model. Bhuiyan, Greer, and McCalla (1990) also report that all 6 participants in their study exhibited the looping model immediately after their first lecture on recursion. Fortunately, the looping model usually becomes less frequent as participants receive more training. The 6 participants in Bhuiyan et al.'s (1990) study started to abandon the looping model two weeks after the first interview. However, a minority of participants persistently exhibit this model even after substantial training. Sanders et al. (2006) and Scholtz and Sanders (2010) report that an average of 8% (ranging from 0% to 19%) of participants from seven different cohorts (average n=136) exhibited the looping model in evaluating various recursive functions in tests near or at the end of semester. Mirolo (2010) reports that less than an average of 8% (ranging from 0% to 20%)[2] of participants from two cohorts (n_1=45, n_2=50) exhibited the looping model in evaluating two different recursive functions in final exams.

[2] Mirolo (2010) creates a category called *other* models to include the looping model and several other models and reports that 0%, 2%, 10%, 20% of the participants were categorized as holding other models.

Active Model

Model profile. The *active* model (Götschi et al., 2003), also called the *forward* model (Mirolo, 2010), is a slightly incomplete view of recursive function. In this model, recursive calls do trigger new instantiations and suspend unfinished operations. The newly generated parameter values are used in the new instantiations rather than overriding the previous ones. The active flow is carried out without any problem. However, once the base case is reached, the result is obtained by finishing suspended operations all at once without a passing return through each instantiation (Götschi, 2003). Figure 3 illustrates a trace representing the active model.

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

Figure 3. An active trace of the factorial function with an input of 3.

Associated misconceptions. There are two explanations for the active model (Götschi, 2003; Götschi et al., 2003; Sanders et al., 2006). First, students who exhibit the active model may actually understand the recursive process but do not show the trace representing the passive flow because some recursive functions simply pass returned values to previous instantiation without processing the values. Second, the students do not understand the passive flow. However, an examination of the sample trace representing the active model (Götschi, 2003) shows that the students mistakenly processed operations in the instantiated head-blocks before the base case is reached. Although this mistake does not lead to a wrong answer, it indicates a misconception of the execution mechanism—specifically the suspension of operations for delayed processing. Thus, if the sample trace does represent students' performance, the possibility that students understand the recursive process should be disregarded.

Frequency and persistence. The active model is typically found only in students' evaluation performance near or at the end of a course. However, it appears to be more persistent than the looping model. Sanders et al. (2006) and Scholtz and Sanders (2010) report that varying proportions of participants from the seven cohorts (average 20%, ranging from 3% to 48%) exhibited the active model in evaluating different types of recursive functions. Similarly, Mirolo (2010) reports that a varying proportion of participants from the two cohorts (average 4%, ranging from 0% to 11%) exhibited the active model in evaluating two different recursive functions.

Copies Model

Model profile. The c*opies* model of recursive function, which characterizes expert knowledge of recursion, refers to a view of recursive function as "a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones" (Kahney, 1983, p. 235). Other researchers use synonymously the term *stack* model (Bhuiyan et al., 1990) or *sound* model (Mirolo, 2010). According to Götschi's (2003) coding scheme, students' traces must demonstrate these three features: copies of function instantiations at the active flow, a switch of flow direction at the base case, and the copies of function instantiations at the passive flow. Figure 4 shows a trace of factorial function that exemplifies the copies model:

```
factorial (3)
3*     factorial (2)
3*     2*    factorial (1)
3*     2*    1*      factorial (0)
3*     2*    1*      return: 1
3*     2*    return: 1
3*     return: 2
return: 6
```

Figure 4. A copies trace of the factorial function with an input of 3

Frequency and persistence. Students hardly construct the copies model immediately after introduction of recursion. Kahney (1983) tests students with a function evaluation task immediately after an introduction of recursion. Only 1 of the 30 participants developed the copies model as shown in his/her responses. Many students are able to develop this model after substantial training. Sanders et al. (2006) and Scholtz and Sanders (2010), for example, show that a large but varying proportion of participants (average 51%, ranging from 26% to 84%) from the seven cohorts in their studies exhibited the copies model. Similarly, Mirolo (2010) reports that 58% to 76% of participants from two other cohorts exhibited the copies model. Unfortunately, the copies model exhibited in students' evaluation traces may not be generalized to full mastery of the copies model. Scholtz and Sanders (2010) demonstrate that while many students (28% to 55%) exhibited the copies model in their evaluation traces, only a few of them (8%) were able to describe in plain language the execution process of a given function, and none of them correctly described the general execution mechanism of recursive functions.

Other Models

Several other mental models also exist in small proportions. The *step* model (Götschi et al., 2003), also called the *do-it-once-more* model (diCheva & Close, 1996), involves a misconception that the recursive call triggers execution of the head-block for only one more time. The *return value* model involves the misconception that every instantiation returns a value and these values are later combined into the solution (Götschi et al., 2003; Sanders et al., 2006). A *magic or syntactic* model refers to the interpretation of recursive functions with some ideas of the syntactic elements indicative of recursive behavior but not a clear view of how the function accomplishes a goal (Götschi et al., 2003; Kahney, 1983). The *algebraic* model refers to interpretation of recursive function as algebraic problems (Götschi et al., 2003; Kahney, 1983). The *odd* model refers to interpretation of recursive function with idiosyncratic ideas about some features of recursive functions (Götschi et al., 2003; Kahney, 1983). Lastly, the *null* model refers to the belief that a procedure cannot be used within itself, and the computer will reject the procedure (Kahney, 1983). Because these mental models are attributable to misconceptions of prerequisite concepts such as the value return mechanism or misconceptions of general computational process, they will not be the focus in this study.

Stability of Mental Models of Recursion across Tasks

Many studies have found that mental models and associated misconceptions manifest themselves inconsistently across task situations. Götschi (2003), for example, reports that 59% of the participants (n = 169) exhibited different mental models in evaluating different recursive functions. Sanders et al. (2006) also report that half of the participants who exhibited the copies model (n_{copies} = 95) in one question moved back to the active model or even the looping model in another question. A study with younger students (10 to 14 years old) shows that some participants even exhibited different mental models in evaluating very similar functions. Further, several studies indicate that percentages of students categorized as exhibiting the copies model vary by the type of functions presented (Mirolo, 2010; Scholtz & Sanders, 2010; Segal, 1995). In addition, inconsistency may even occur within a single reasoning episode as shown in Segal's (1995) analysis of one participant's think-aloud protocol.

Despite the prevalence of this phenomenon, the literature is limited in analyzing the underlying factors and explaining the mechanism of influence. An extended analysis of the literature that reveals multiple potential factors including task requirement, task order, function complexity, parameter type and call structure. For each factor, existing explanations are reviewed and critiqued or elaborated in the sections below.

Effect of Task Requirement

Task requirement refers to specific actions that students are required to perform in a task. To elicit students' knowledge of recursive function, a test may require them to trace a function with given inputs (e.g., Götschi et al., 2003), predict outputs of a function with given inputs (the students may or may not trace the function; e.g., Kahney, 1983), evaluate the correctness of a function without tracing it (e.g., Kahney, 1983), describe in natural language the execution process of a specific recursive function (e.g., Scholtz & Sanders, 2010), or describe in natural language the general execution mechanism of recursive function (e.g., Scholtz & Sanders, 2010).

Different task requirements appear to influence the manifestation of mental models. For example, George (2000b) reports that students were much more likely to demonstrate the copies model when they were allowed to trace function execution using diagrams than when they were required to mentally evaluate the functions. Many students admitted that they mostly relied on tracing to understand recursive functions. In the same vein, Scholtz and Sanders (2010) report that while a majority of students (45% to 68% for three functions, n=123) exhibited the copies model in tracing recursive functions, only a few (8% of a subsample, n=118) exhibited the copies model when asked to describe in plain language the execution process of a specific function. Moreover, none of them showed solid mastery of the copies model when asked to describe in plain language the general execution mechanism of recursive function. They either did not mention the passive flow in their responses or incorrectly stated that execution would be terminated at the base case.

Despite its significant influences on performance, there is little theory to explain task requirement effects. George (2000b) simply concludes that students need to use diagrammatic trace to aid their evaluation of recursive functions. Scholtz and Sanders (2010) argue that students use the tracing method in a mechanical manner without real understanding of recursive function. However, these conclusions contribute little to understanding of the phenomenon.

Effect of Task Order

Order of task presentation also appears to be influential. In Segal's (1995) study, none of the participants who evaluated the number function first (n=17) exhibited the base-case-as-stopping misconception, whereas 25% of the participants who evaluated the string function first (n=16) did. Segal explains that the number-first participants developed a sound strategy in evaluating the number function and then transferred the strategy in evaluating other functions, whereas the string-first participants developed a unsound strategy and then transferred to other functions.

Effect of Function Complexity

Function complexity refers to degree of complexity due to the quantity and configuration of components in a function. Data presented in several studies suggest that function complexity accounts for the variance in mental model manifestation. For example, participants in Mirolo's (2010) study were more likely to exhibit the copies model when evaluating a linear function (67 of 88 participants) than evaluating a tree function (49 of the same 88 participants). The linear function involved two number parameters and two recursive calls—one tail call and one prefix-only call. The tree function, more complex than the linear function, involved two string parameters and three recursive calls—one tail call and two mutually tied calls. Similarly, Scholtz and Sanders (2010) found that the copies model manifested more frequently for a list-multiplication function (64 of 123 participants) and a list-summation function (68 of the same 123 participants) than for a sum-of-list-powers function (35 of the same 123 participants). The list-multiplication and the list-summation functions both only contain one head-block-only call, while the sum-of-list-powers function involves two recursive sub-functions, one of which contains a prefix-only call and the other contains two mutually tied calls.

As shown in Table 1, these data suggest that the more complex the function, the less likely the participants are to exhibit the copies model. However, the functions paired for comparison differ in multiple task features. For example, the linear function and the tree function (Mirolo, 2010) differ in parameter type, number of recursive calls, and call structure. Thus, it is difficult to isolate any single task feature as solely responsible for the manifestation of mental models.

Table 1
Task features of the recursive functions used to test participants in two previous studies (Mirolo, 2010; Scholtz & Sanders, 2010)

References	Function 1	Function 2
Mirolo (2010) n=88	“linear” 2 number parameters 1 tail call 1 prefix-only call	“tree” 2 string parameters 1 tail call 2 mutually tied calls
Scholtz & Sanders (2010) n=123	“list multiplication” “list summation” 1 list parameter 1 head-block-only call	“sum of list powers” 2 sub-functions 1 prefix-only call 2 mutually tied calls
Function complexity	Relatively low	Relatively high

Note. In both studies (Mirolo, 2010; Scholtz & Sanders, 2010), function 2 appears to be more complex than function 1 on the basis of parameter type, number of recursive calls, and call structure.

When participants do not exhibit the copies model consistently across tasks, they are most likely to use the active model or the looping model. Götschi et al. (2003) and Sanders et al. (2006) show that of participants exhibiting the copies model when evaluating a recurrence relation function, only 37% to 48% continued to apply it when evaluating a list-manipulation function, 13% to 35% changed to the active model, 17% to 33% to the looping model, and 0% to 17% to other models (e.g., magic, odd, etc.). Thus, students may hold multiple mental models and select the best match for the task at hand, even if the models themselves entail substantial misunderstandings of the execution mechanism (Götschi et al., 2003; Sanders et al., 2006).

The role of individual task features, however, may provide an alternative explanation. The recurrence-relation function and the list-manipulation function differ in two ways. First, the former has a number parameter, while the latter has a list parameter. Second, the former has a *prioritized prefix-and-suffix* call, while the latter has a *head-and-prefix* call. The prioritized prefix-and-suffix number function appears to predominately induce the copies model, whereas the head-and-prefix list function appears to induce various mental models without an emphasis.

These particular task feature configurations, however, complicate the function complexity hypothesis. The list parameter is more complex than the number parameter due to number of elements involved. The prioritized prefix-and-suffix call is more complex than head-and-prefix call in terms of structural configuration. Thus, the two functions have task features with opposing degrees of complexity (see Table 2). In accordance with the function complexity hypothesis, the parameter type may predominately affect the manifestation of mental models, overriding the effect of call structure. Alternatively, other mechanisms may govern this effect, particularly when function complexity is below certain threshold.

Table 2
Mental model manifestation in evaluating two recursive functions

References	Recurrence-relation function		List-manipulation function
Götschi et al. (2003) n_{copies}=77	100% copies model		37% copies model 13% active model 33% looping model 17% other models
Sanders et al. (2006) n_{copies}=95	100% copies model		48% copies model 35% active model 17% looping model 0% other models
Function complexity	1 number parameter	<	1 list parameter
	1 prioritized prefix-and-suffix call	>	1 head-and-prefix call

Note. Participants who exhibited the copies model when evaluating the recurrence-relation function shifted to other models when evaluating the list-manipulation function (Götschi et al., 2003; Sanders et al., 2006).

Effect of Parameter Type

Parameter type refers to the general data type (e.g., number, list, and string) used for the parameters of a function. A majority of students correctly evaluate number functions while a minority evaluates list functions correctly (Götschi, 2003). Götschi offers two explanations: (1) lists are more difficult to manipulate than numbers because lists contain multiple elements; (2) extra list-related commands add complexity to evaluating recursive functions. However, examination of the study data reflects differences between the number and list functions beyond parameter type, introducing the possibility that other factors contribute to the selection and application of the problem solving strategy (see Table 3).

Table 3
Task features of the recursive functions used to test participants in Götschi's (2003) study

Functions	Percentage of participants who correctly evaluate the function	Task Features
Number function 1	72% to 77%	1 number parameter 1 head-block-only call
Number function 2	45% to 62%	1 number parameter 2 mutually tied calls
List function 1	25% to 27%	2 sub-functions 1 list parameter 1 number parameter 1 nested call
List function 2	25% to 36%	1 list parameter 1 embedded & prioritized call

Note. Task features of the recursive functions used to test participants in Götschi's (2003) study vary by multiple dimensions including parameter type, number of parameters, number of recursive calls and call structure.

Only one study examines the effect of parameter type with these potential confounding factors controlled. In this study (Segal, 1995), 33 participants evaluated three structurally similar embedded recursive functions with different parameter types (i.e., number, list, and string). Sixty-four percent of them correctly evaluated the number function, whereas only 36% and 39% correctly evaluated the list function and string function, respectively. Participants' correct evaluation indicates that they exhibited the copies model, because only the copies model is productive for evaluating embedded recursion.

Segal (1995) explains that in familiar domains such as number, students are able to contemplate unprocessed operations in instantiated recursive cases in their entirety. This holistic view enables students to employ a sound evaluation strategy, which delays processing of the operations until all recursive cases are fully instantiated. In unfamiliar domains such as list and string, students have not developed this holistic view of unprocessed operations, thus they tend to employ an unsound evaluation strategy which immediately processes operations before instantiation of all recursive cases.

Although Segal (1995) did not publish the raw data (i.e., a cross-tabulation of parameter types and evaluation strategies) that might support her view, her domain familiarity hypothesis is plausible given the substantial body of mathematics education literature positing that through learning students become able to contemplate elements and operations in a mathematical process as a single mathematical object (Sfard, 1991).

Effect of Call Structure

Call structure refers to the configuration of a recursive call in terms of prefix, suffix, head-block, tail-bock, and other features. No study has systematically examined the effect of call structure, although data presented by Sanders et al. (2006) suggest an effect of call structure on the manifestation of mental models. In that study, two cohorts of participants evaluated the list-manipulation function and two others evaluated the list-calculation function in their class tests. Assuming that the four cohorts are equivalent samples, the data suggest that substantially more participants exhibited the copies model in evaluating the calculation function than in the manipulation function. Many participants regress from the copies model to the looping or active model when the task switches from the former function to the latter function (see Table 4).

Table 4

Mental model manifestation in evaluating two recursive functions among four cohorts of participants (Sanders et al., 2006)

	Cohort Yr	N	Copies model	Looping model	Active model	Other models
List Manipulation (head-and-prefix call)	2002	139	26%	14%	25%	35%
	2003	127	37%	13%	31%	20%
List Calculation (prioritized prefix-and-suffix call)	2004	153	73%	0%	5%	22%
	2005	101	64%	0%	10%	26%

Note. Assuming that the four cohorts are equivalent samples, participants are more likely to exhibit the copies model when evaluating the list-manipulation function than when evaluating the list-calculation function.

The only difference between the two functions is the call structure. The list-manipulation function has a head-and-prefix call, while the list-calculation function has a prioritized prefix-and-suffix call in which the prefix operations have priority over the suffix operations. The prioritized prefix-and-suffix call appears to predominately induce the copies model, whereas head-and-prefix call appears to induce various mental models without an emphasis (under the assumption that the participant cohorts are equivalent).

The puzzling point of this result is that the seemingly more complex function (i.e., function with a prioritized prefix-and-suffix call) appears easier to evaluate than the seemingly less complex function (i.e. function with a head-and-prefix call). This result, however, clarifies the finding discussed in the previous section. That is, the number function with a prioritized prefix-and-suffix call (i.e. the recurrence-relation function) is more likely to induce the copies model than the list function with a head-and-prefix call (i.e. the list-manipulation function). The counterintuitive effect of call structure identified here suggests that both task features may influence mental model selection similarly.

Reconsidering Mental Model

In summary, existing explanations for the instability of mental models do not account for the available data related to learning recursion, and no single explanation exists to accommodate the effects of various factors. This lack of a parsimonious explanation suggests a need to reconsider the nature of mental models as applied in CS domains. Specifically, misconceptions—the assumed components of maladaptive mental models—may not exist as unitary and stable mental structures. Instead, their manifestation may be explained by other underlying mental structures or processes.

Knowledge in Pieces Theory

The knowledge-in-pieces theory offers an alternative account of conceptual change (diSessa, 1983; 1988; 1993). The contemporary dominant view holds that conceptual change occurs when misconceptions[3] are confronted and replaced (e.g., Clement, 1982; McCloskey, 1983). In contrast, the knowledge-in-pieces theory posits that conceptual change occurs when small, isolated knowledge elements, which underlie the observed misconceptions, are organized into knowledge clusters that are productive in problem solving (diSessa, 1993).

One learning phenomenon explained by the knowledge-in-pieces theory is that

[3] Also known as *preconceptions* (Clement, 1982; Glaser & Bassok, 1989), *alternative conceptions* (Hewson & Hewson, 1984), *alternative frameworks* (Driver & Easley, 1978), and *naive theories* (McCloskey, 1983; Resnick, 1983).

novices' interpretations of physical events are often inconsistent across situations governed by the same principle (diSessa, 1993). This phenomenon is analogous to the observations made above on the mental models of recursion. That is, task features such as the call structure of recursive function appear to have a systematic influence over which mental model introductory CS students may exhibit. Thus, the knowledge-in-pieces theory may shed light on this unexplained phenomenon in learning recursion.

According to the knowledge-in-pieces theory (diSessa, 1993), intuitive knowledge of mechanics comprises of a large number of phenomological primitives (p-prims), which are small, intuitive knowledge elements derived from experiences of common physical phenomena. A p-prim is activated by its designated contextual configuration, which is specified by previously activated knowledge elements such as sensory schemata or other p-prims. The likelihood of a p-prim to be activated by a contextual configuration is referred to as *cuing priority*. P-prims are self-explanatory, affording explanation and prediction of physical phenomena without the need to be further justified. Thus, students can quickly generate intuitive interpretations of a given physical event. For example, when people see someone kicking a ball, they can immediately predict the ball's motion due to activation of a p-prim called "force as a mover" (diSessa, 1993, p.129).

In a naïve knowledge system, p-prims are the main components, and *symbolic schemas* (e.g., scientific concepts and principles) are largely absent. Each p-prim is responsible for its own specific contextual configuration, and activation of one p-prim rarely spreads to others. Even if activation spreading does occur, it usually remains local and revolves around the initial p-prim due to its high *reliability priority*. Thus, a slightly changed situation may activate a different p-prim or a few clustered p-prims, resulting in an interpretation inconsistent with the previous one. For example, when people see the ball slowing down, they explain that all moving objects eventually stop due to activation of a p-prim called “dying away” (diSessa, 1993, p.133) and fail to consider the notion of force which applies to both the initiation of the ball’s motion and its return to a state of rest. In sum, different problem situations give rise to different contextual configurations which, in turn, activate different p-prims. As a result, students form discrepant interpretations of conceptually equivalent scenarios.

As people learn physics, symbolic schemas (e.g., Newton’s laws) gradually enter their knowledge systems through explicit learning and interact with p-prims to generate complex mental processes (Hammer, Elby, Scherr, & Redish, 2005; Thaden-Koch, Dufresne, & Mestre, 2006; diSessa & Sherin, 1998). Ideally, productive symbolic schemas should be directly activated in their designated circumstances to generate scientific solutions. However, p-prims are always activated first due to their high cuing priorities. The activated p-prims change the original context and reprioritize knowledge elements in the system. In some cases, the original context is malleable to the extent that the active p-prims substantially change it to favor maladaptive schemas or other p-prims. In other cases, the original context is robust enough to limit the influence of the active p-prims and to maintain the high priority of the normative schemas.

For example, placing a hand on a spring evokes the Ohm’s p-prim, which refers to “an agent or causal impetus acts through a resistance or interference to produce a result” (diSessa, 1993, p.217). The activated Ohm’s p-prim necessitates the source of resistance, a contextual configuration that easily cues the springiness p-prim in this particular hand-on-spring situation. The springiness p-prim, or “deformation with consequent development of restoring force" (p.134), then justifies the existence of force from the object, a contextual configuration that cues the Newton's third law (Clement, 1993). In contrast, pushing a rigid and heavy object also evokes the Ohm's p-prim. However, in this rigid-and-heavy-object situation, the Ohm's p-prim cues the intrinsic resistance p-prim instead, which terminates the interpretation without reference to springiness or Newton's third law (diSessa, 1993). In short, different problem situations shape p-prim-generated contextual configurations differently so that they cue different knowledge elements and produce different interpretations.

Compared to novices, experts often interpret physical events consistently using the underlying scientific principles. In an expert knowledge system, activation of a p-prim rapidly and consistently spreads to a cluster of knowledge elements, a coordination class, which manifests formal knowledge including scientific principles (diSessa & Sherin, 1998). Activation of the formal knowledge in turn shapes the contextual configurations of the initial p-prim and other knowledge elements in the activation sequence, changing their functions in the cognitive process. P-prims in a well-developed coordination class, for instance, only serve as heuristics cues to formal knowledge instead of affording interpretations independently. The formal knowledge orients both conceptual and perceptual schemas to extract the essential information from a situation. Thus, superficial differences between problem situations do not influence experts' interpretations as they do to novices'.

In summary, the knowledge-in-pieces theory provides plausible explanations for novices' inconsistent interpretations across problem situations. These explanations may inform the investigation of the similar phenomenon in learning of recursion. That is, students often exhibit different mental models of recursion in superficially different problem situations.

A Possible P-Prim: Re-analysis of Existing Data

Preliminary analysis of existing data demonstrates the utility of the knowledge-in-pieces theory and indicates the initial profile of a possible p-prim. Three main attributes of p-prims are used in the analysis per diSessa (1993): First, p-prims are self-explanatory, thus a sense of intuitive obviousness and satisfaction should exhibit in students' think-aloud or interview data. Second, p-prims are functional, thus they should serve the students well in many cases rather than causing problems all the time. Third, p-prims are sensitive to contextual configuration, thus when students change ideas there should be corresponding contextual change. In relation to the third attribute, features of a task shape p-prim-generated contextual configurations, which in turn reinforce or suppress the p-prim's activation. To summarize this effect of task features on the operation of p-prim, I propose a construct called *situational constraint* to refer to the extent to which task situations prevent a p-prim from operating in inappropriate circumstances.

Erratic Performance within a Single Task

The first data source is drawn from think-aloud protocols presented by Segal (1995). One participant evaluated a function which takes any numeric input *n* to generate (2*n -1) lines of stars, with the number of stars on each line decreasing from n to 1 and then increasing back to n. This function is written:

```
hn = stars1++“\n”, if n=1
   = starsn++“\n”++h(n-1)++starsn++“\n”, otherwise
```

In this function, *h* denotes the function name, n denotes the parameter, *stars*n means to print n number of stars, ++ means to append lists, and “\n” means to generate a new line.

The correct evaluation and output of *h*3 using the substitution method is:

```
h3 = stars3++“\n”++h(2)++stars3 ++ “\n”
   = stars3++“\n”++(stars2++“\n”++h1++stars2++“\n”)
     ++stars3++“\n”
   = stars3++“\n”++(stars2++“\n”++(stars1++“\n”)
     ++stars2++“\n”)++stars3++“\n”
***
**
*
**
***
```

The participant, however, gave an incorrect answer, which generated:

```
***
**
*
```

Segal (1995) attributes this response to the participant's misconception that the base case is the stopping condition for the recursive function. She also notes that this misconception manifests inconsistently during the participant's thinking process. When the participant instantiates the first recursive call, he appears to understand that the suspended operations should be carried out after evaluation of the base case. However, when he eventually completes the evaluation of the base case, he unexpectedly abandons those suspended operations. Moreover, this mistake is not due to carelessness, as he double-checks and confirms his evaluation process.

Further analysis, however, reveals that operation of an intuitive idea may account for the inconsistent manifestation of the misconception. Figure 5 presents each idea that the participant voiced during the think-aloud session with the corresponding elements in his trace and output. This representation of the data segments the think-aloud protocol into structurally and functionally similar units. Each unit contains an external stimulus, an idea associated with the stimulus, and a result generated by the operation of the idea.

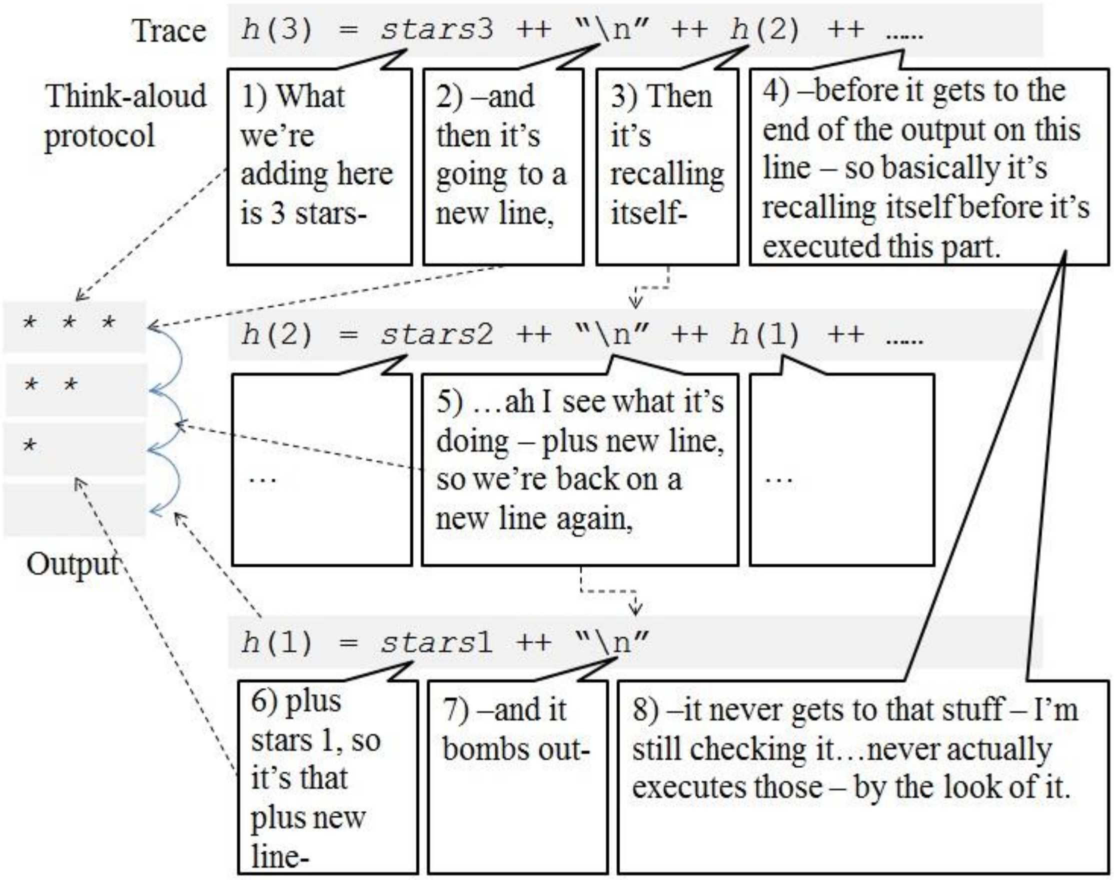

Trace
h(3) = stars3 ++ "\n" ++ h(2) ++ ......
Think-aloud protocol
1) What we're adding here is 3 stars-
2) –and then it's going to a new line,
3) Then it's recalling itself-
4) –before it gets to the end of the output on this line – so basically it's recalling itself before it's executed this part.
* * *
* *
*
Output
h(2) = stars2 ++ "\n" ++ h(1) ++ ......
…
5) …ah I see what it's doing – plus new line, so we're back on a new line again,
…
h(1) = stars1 ++ "\n"
6) plus stars 1, so it's that plus new line-
7) –and it bombs out-
8) –it never gets to that stuff – I'm still checking it…never actually executes those – by the look of it.

Figure 5. Representation of participant TS's think-aloud protocol presented in Segal's (1995, p.398) study

This participant's thinking process can be further simplified by extracting the essence of each idea. As shown in Figure 6, every idea can be simplified to "(the computer will) do this (and the result is…)." This idea appears to be self-explanatory to the participant as he smoothly traces through all the programming operations without rationalizing or explaining them. Particularly, when he reaches the base case, he instantly asserts that "it bombs out" (Segal, 1995, p.399). Nevertheless, this idea is largely functional as the participant correctly evaluates most of the operations. Were the function a tail-recursion, he might have reached the correct answer using this idea. Further, this idea is sensitive to context. The participant initially decides that the computer would carry out the suspended operations but soon abandons this idea and even considers the suspended operations as a “red herring” (Segal, 1995, p.399) according to a follow-up interview. It appears that the base-case assertion changes the contextual configuration of the suspended operations. As a result, the “(the computer will) do this (and the result is…)” idea deactivates and the “red herring” schema activates and supplies a plausible explanation for the suspended operations.

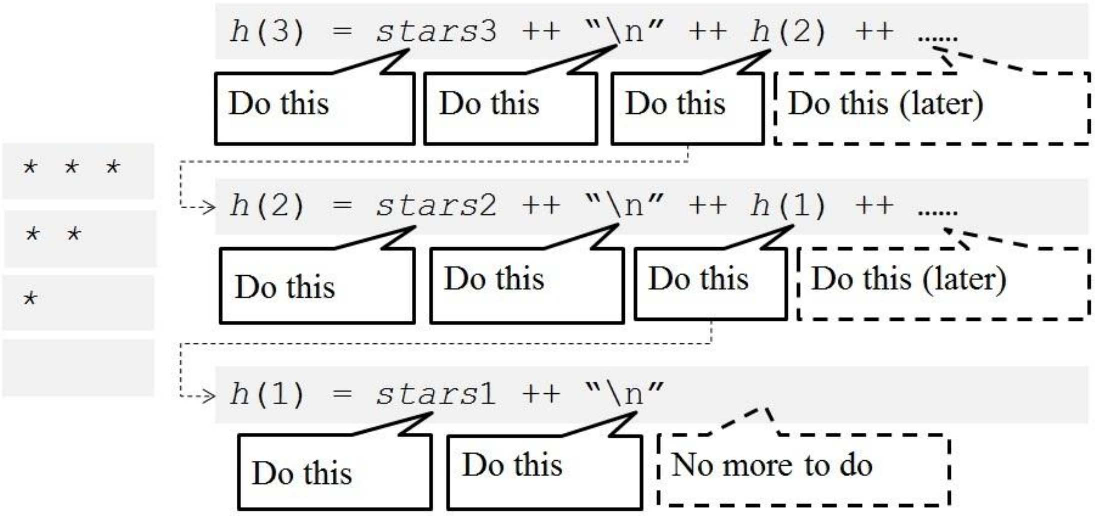


Figure 6. Participant TS's thinking process (Segal, 1995, p.398) simplified by extracting the essence of each idea

All the characteristics indicated above point to the possibility that the “do this” idea is a p-prim, which may offer an alternative explanation for the participant’s erratic performance. That is, the “do this” idea is activated by the contextual configuration that comprises symbols representing programming operations. No exception occurs even when he reaches the suspended operations on the first trace line, as he clearly notes that the computer will carry out those operations later. This particular contextual configuration (i.e., symbols representing programming operations) vanishes as no further operation is present after the last operation (i.e., “\n”). Thus, the participant decides that the computer has no more to do. His reasoning is a product of consistently and sequentially applying the “do this" idea to each individual operation and appears to be flawless from his point of view. Therefore, it is unnecessary to assume that the participant has a misconception but applies it inconsistently.

The characteristics of this function also contribute to the participant's erratic performance. In this particular problem situation, all individual operations (e.g., print three stars or generate a new line) can be processed without the result returned by instantiation of the recursive call. This characteristic imposes a very low situational constraint to the "do this" idea, allowing it to freely function in inappropriate circumstances. If the operation adjacent to the recursive call cannot be processed without the result returned by the recursive call (e.g., operation 3+recursive call), application of "do this" to the operation will be constrained until all recursive instantiations are evaluated. In sum, it is plausible that the "do this" idea is a p-prim and operates differently in situations with different levels of constraint.

Erratic Performance across Tasks

In the above review of factors influencing the manifestation of mental models, a puzzling phenomenon emerges: the seemingly more complex functions are more likely to induce the correct copies model than the seemingly less complex functions (e.g., Sanders et al., 2006). However, when reformulated in terms of the "do this" idea, behavior changes in response to different levels of situational constraint, providing a coherent account.

Effect of Call Structure

Students are more likely to exhibit the copies model in a list-calculation function than in a list-manipulation function (Sanders et al., 2006). An examination of the call structure of the two functions reveals that the list-calculation function has higher situational constraint to the “do this” idea than the list-manipulation function. As shown in Table 5, the list-calculation function (P_{LC}) has a prioritized prefix-and-suffix call. Both the prefix operation “head(list) +” and the suffix operation “(…)*3” (p.141) cannot be carried out until the result of the recursive call is obtained. Thus, no processing would occur at each instantiation during the active flow. All of the operations must be processed through the passive flow of recursive process. Such a task situation inhibits the “do this” idea from taking effect at the wrong time. As a result, students generate correct results and exhibit the copies model.

In contrast, the list-manipulation function (P_{LM}) has a relatively low situational constraint to the “do this” idea. It has a head-and-prefix call. “||” is the prefix operation which cannot be carried out until the result of the recursive call is obtained. “2*head(list)” is the head operation which can be carried out before instantiating the recursive call. During evaluation, operation “||” can also become executable due to processing of the head operation. The particular structure of this function loosens its situational constraint to the “do this” idea. For instance, the mathematical operation “2*4” can be immediately carried out to generate the result of “8”, and the list element binding operation “8||2” can be immediately carried out to generate the result of list “8 2”. Such behaviors are clearly shown in traces which exemplify the looping model (see Table 5).

Table 5

Comparison of situational constraint between two recursive functions

<table>
<tr><th>list-calculation function</th><th>list-manipulation function</th></tr>
<tr><td>P_{LC}(list)
 if list is empty
  then, return 1
 else
  (head(list) + P_{LC}(tail(list)))*3</td><td>P_{LM}(list)
 if list is empty
  then, return 1
 else
  2*head(list) || P_{LM}(tail(list))</td></tr>
<tr><td>Looping trace:
P_{LC}(2 1 3 1)
=(2+P_{LC}(1 3 1))*3
[impossible to carry out any operation]</td><td>Looping trace
P_{LM}(4 1 3 5) [carry out the following]
=(8 P_{LM}(1 3 5)) [“2*4”]
=(8 2 P_{LM}(3 5)) [“2*1”; “8||2”]
=(8 2 6 P_{LM}(5)) [“2*3”, “8 2||6”]
=(8 2 6 10 P_{LM}( ))[“2*3”, “8 2 6||10”]
=(8 2 6 10 1) [“8 2 6 10||1”]</td></tr>
<tr><td>Copies trace:
P_{LC}(2 1 3 1)
=(2+P_{LC}(1 3 1))*3
=(2+(1+P_{LC}(3 1))*3)*3
=(2+(1+(3+P_{LC}(1))*3)*3)*3
=(2+(1+(3+(1+P_{LC}())*3)*3)*3)*3
=(2+(1+(3+(1+1)*3)*3)*3)*3
=(2+(1+(3+2*3)*3)*3)*3
=(2+(1+(3+6)*3)*3)*3
=(2+(1+9*3)*3)*3
=(2+(1+27)*3)*3
=(2+28*3)*3
=(2+84)*3
=86*3
=258</td><td>Copies trace:
P_{LM}(4 1 3 5)
=2*4||P_{LM}(1 3 5)
=2*4||(2*1||P_{LM}(3 5))
=2*4||(2*1||(2*3||P_{LM}(5)))
=2*4||(2*1||(2*3||(2*5||P_{LM}())))
=2*4||(2*1||(2*3||(2*5||1)))
=2*4||(2*1||(2*3||(10||1)))
=2*4||(2*1||(2*3||(10 1)))
=2*4||(2*1||(6||(10 1)))
=2*4||(2*1||(6 10 1))
=2*4||(2||(6 10 1))
=2*4||(2 6 10 1)
=8 2 6 10 1</td></tr>
</table>

Note. The list-calculation function imposes higher situational constraint to the “do this” idea than the list-manipulation function does.

Effect of Parameter Type

Students perform better on a number function than on a structurally similar list function (Segal, 1995). Segal hypothesizes that students more effectively contemplate unprocessed operations in recursive instantiations as an entirety within familiar domains (i.e., numbers) than in unfamiliar domains such as lists. Such ability leads the students to employ the sound evaluation strategy, which delays processing of the operations until all recursive cases are fully instantiated.

Segal’s (1995) domain familiarity hypothesis can be rephrased in terms of situational constraint. That is, the situational constraint to the “do this” idea is imposed by students’ prior knowledge of the relevant domain. If students are familiar with the domain, then it is likely they are able to view a process involving unprocessed operations as an entire object. Thus, they are comfortable leaving the unprocessed operations in recursive instantiations as an entirety instead of attempting to process them. Such tendencies inhibit the “do this” idea from operating at the wrong time. If students are unfamiliar with a domain, however, they tend to process the individual operations in recursive instantiations as soon as they can, due to their inability to contemplate them as an entire object. As a result, the “do this” idea freely operates regardless of the timing.

Effect of Task Order

Students who first evaluate a number function outperform those who first evaluate a string function (Segal, 1995). Given that the students evaluate the two functions consecutively, it is possible that the high situational constraint in the number function primes the number-first students (Tulving & Schacter, 1990) and raises the situational constraint in the string function. The string-first students, primed with low situational constraint in the string function, apply the “do this” idea more freely in evaluating the number function.

Effect of Task Requirement

Students are more likely to exhibit the copies model when they are allowed to trace recursive functions than when they are asked to mentally evaluate the functions (George, 2000b) or describe in plain language the execution process (Scholtz & Sanders, 2010). Using external representation, students are able to externalize and maintain the intermediate products generated by their mental operations. These intermediate products may in turn shape the contextual configuration and constrain the operation of the "do this" idea.

For example, students may use diagrams to represent call instantiations and flow of control (e.g., Troy & Early, 1992). The diagrams will maintain an overview of the execution process and refrain students from applying the "do this" idea at the wrong time. Therefore, it is possible that tasks requiring the usage of external representation impose a higher situational constraint than tasks that do not require or permit external representation.

A Possible Instance of an Agentive Causality Meta-P-Prim

A meta-p-prim is the abstraction of many related p-prims. For example, many p-prims in the domain of mechanics share the same causal syntax: an *agent* takes an *action* on a *patient* to generate certain *effect* (diSessa, 1993). The Ohm's p-prim involves some agent exerting a variable effort against a variable resistance to generate a variable result. The closely related "force as a mover" p-prim involves some agent exerts a directed violent impetus on some object at rest to move it in a certain direction. In fact, the natural language is largely framed in this syntax. In the sentence "a boy throws a rock into a pond", the "boy" agent takes the "throw" action on the "rock" patient to generate the "into the pond" effect. The four components—agent, action, patient, and effect—are always activated together. If a situation only offers information for some components, the mind instantly generates possible contents for other components. In the "a man kicks a ball" situation, with the agent, action and patient known, the mind instantly proposes the effect "the ball will be rolling".

The "(the computer will) do this (and the result is…)" idea has a syntactic structure similar to that of the *agentive causality* meta-p-prim. The "(the computer will) do this (and the result is…)" idea has the same four components and relations among the components.

It is possible that this structural similarity is grounded in similar mechanisms. Although reasoning in programming does not seem to rely on agentive experiences as reasoning in physics does, it is reasonable to expect the existence of agentive thinking, because the computer is often conceptualized as a surrogate of a human problem solver that processes inputs to generate outputs on behalf of the programmer (e.g., Pea, Soloway, & Spohrer, 1987).

Although programming is notational and physics is physical, there is no inherent cognitive difference between a notational object and a physical object (von Glasersfeld, 1995). In mathematics education research, for example, it is widely recognized that mathematical concepts are object-like and subject to mental manipulations (e.g., Sfard, 1991). Given the representational existence of the agent and the patient (i.e., object) in the domain of programming, it is possible that the "do this" idea is an instance of agentive causality in understanding of recursive function.

Research Questions

As posed in Chapter 1, the research questions of this study are:

1. Do beginning CS students demonstrate reliance on identifiable p-prims when trying to understand and apply recursion?
2. If they do, what are the structures, relevant circumstances, functions, and effects of these p-prims as they impact learning and performance?

As a consequence of the above analyses, it is reasonable to suspect that there exists a p-prim in beginning CS students' understanding of recursive function, which could be referred to as a *computer-as-agent* p-prim: the computer (agent) processes (action) operational instructions (patient) to generate results (effect).

Such a p-prim would function in predictable ways that could be empirically assessed. Patterns of performance on tasks embodying various levels of situational constraint would reflect the varying hypothetical operation of such a p-prim in different conditions. Also, patterns of explicit or implicit interpretation of a presented problem would reveal the specific ways in which such a p-prim operates. These conjectures are specified in the following quantitative hypotheses and qualitative suppositions.

Quantitative Hypotheses

As previously discussed, the higher the situational constraint, the less likely the computer-as-agent p-prim will operate in inappropriate circumstances, leading to appropriate manifestation of the copies model of recursive function. The lower the situational constraint, the more likely the computer-as-agent p-prim will be to operate independently, leading to inappropriate manifestation of other models of recursive function. Extant literature suggests four possible sources of situational constraint: call structure, parameter type, task order, and external representation. In this study, call structure and parameter type are chosen to test the hypotheses because of their controllability.

In a recursive function with a prefix call structure (or suffix, prefix-and-suffix call structure), application of the computer-as-agent p-prim on the prefix operation cannot be completed without the result returned by the recursive call, which leads to natural suspension of the current operation and anticipation of the result return. Thus, the prefix call structure imposes a high situational constraint to the computer-as-agent p-prim, preventing it from operating in inappropriate circumstances. By contrast, in a recursive function with a tail call structure (or head, head-and-tail call structure), application of the computer-as-agent p-prim on the tail operation would be readily completed by generating a local result, not concerning operation suspension at all. Thus, the tail call imposes a low situational constraint to the computer-as-agent p-prim, allowing it to operate in inappropriate circumstances. Accordingly, the first hypothesis is:

Hypothesis 1: Participants are more likely to exhibit the copies model when evaluating a recursive function with a prefix call structure than evaluating a recursive function with a tail call structure.

When evaluating a recursive function with number parameter, participants are able to view a procedure with unprocessed operations as an object-like entirety due to previously acquired mathematical competence. The computer-as-agent p-prim can be applied to the entirety instead of the individual operations. As a result, participants are less likely to process individual operations before all recursive calls are instantiated. Thus, the number parameter imposes a high situational constraint to the computer-as-agent p-prim, preventing it from operating in inappropriate circumstances. By contrast, in a recursive function with a list parameter, the computer-as-agent p-prim cannot be applied to the entire procedure due to unfamiliarity with the domain. Instead, participants will still tend to process individual operations before instantiations of all recursive calls. Thus, the list parameter imposes a low situational constraint to the computer-as-agent p-prim, allowing it to operate in inappropriate circumstances. Accordingly, the second hypothesis is:

Hypothesis 2: Participants are more likely to exhibit the copies model when evaluating a recursive function with a number parameter than evaluating a recursive function with a list parameter.

Among four possible combinations of call structure and parameter type, a recursive function with both a prefix call structure and number parameter would have a high situational constraint. A recursive function with a prefix call structure and list parameter or a tail call structure and number parameter would have a medium situational constraint. A recursive function with a tail call structure and a list parameter would have a low situational constraint. Accordingly, the third hypothesis is:

Hypothesis 3: If participants exhibit the copies model in some but not all task situations that represent four possible combinations of call structure and parameter type, their successful performance will be clustered around tasks with higher situational constraints. Performance patterns representing such trend should occur more frequently than those contradicting it.

Qualitative Suppositions

Above all, the computer-as-agent p-prim must behave in accordance with the main attributes of p-prims proposed by the knowledge-in-pieces theory. Thus, the computer-as-agent p-prim should be self-explanatory, functional, and sensitive to contextual configuration. These expectations are specified in the following suppositions:

Supposition 1: Interpretations generated by the computer-as-agent p-prim are characteristic of a sense of intuitive obviousness and satisfaction.

Supposition 2: Interpretations generated by the computer-as-agent p-prim serve participants well in many cases rather than causing consistent problems.

Supposition 3: Interpretations generated by the computer-as-agent p-prim are sensitive to local context and are likely to change in response to local contextual changes.

Further, the computer-as-agent p-prim should serve different functions through different coordination modes for participants with different levels of understanding of recursive function. Deeper understanding of recursion would be reflected in the stability of their copies model. Thus, it is reasonable to expect:

Supposition 4: Stability of the copies model is associated with coordination mode. The more coordinated the operation of the computer-as-agent p-prim, the more stable the copies model will be.

CHAPTER 3

METHOD

The research questions posed in this study are represented by a series of quantitative hypotheses and qualitative suppositions regarding participants' understanding of recursive function. Moreover, the qualitative suppositions must be validated using data that reflect multiple levels of understanding that are represented through quantifiable performance scores. Thus, a mixed methods design was used. Specifically, an adapted two-phase Explanatory Sequential Design for Participant Selection (Creswell & Plano Clark, 2007) was implemented. In this model, as shown in Figure 7, quantitative data are collected and analyzed first, then the results are used to purposefully sample participants for qualitative data collection and analysis. Lastly, results from both the quantitative phase and qualitative phase are analyzed in an integrated manner.

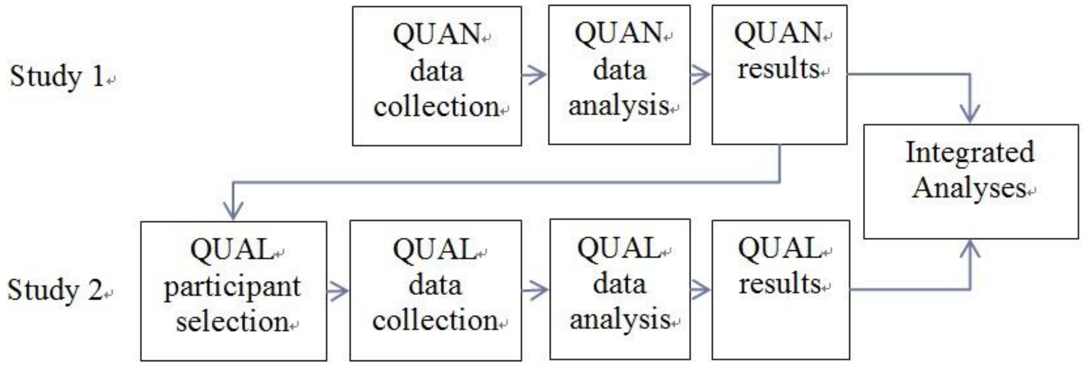


*Figure 7***.** An adapted two-phase Explanatory Sequential Design for Participant Selection (Creswell & Plano Clark, 2007)**.**

The two phases of data collection and initial analyses are organized into two studies. Study 1 aims to test the three quantitative hypotheses previously proposed. Results demonstrating participants' levels of understanding of recursive function were used to purposefully sample participants for Study 2. The second study sought to assesses the three qualitative suppositions. In integrated analyses, assumptions underlying the three quantitative hypotheses were validated using the qualitative data collected in Study 2, and finally results from both studies were taken together to evaluate supposition 4.

Participants and Context

This study was conducted in the fall semester of 2011 in the Introduction to Programming (CS1112) course in the School of Engineering and Applied Science at the University of Virginia. Sixty students were enrolled in the class. Approximately 64% of them were female, and 45% were ethnic minority. Compared to enrollment demographics in other introductory programming courses in the university and nationwide, this course was remarkably inclusive of historically underrepresented populations (Cohoon & Tychonievich, 2011). Seventeen of them were female, and ten were ethnic minority. They majored or intended to major in various disciplines including computer science, biology, engineering, commerce, etc.

All the participants attended 75-minute course meetings on every Monday, Wednesday, and Friday afternoon for 15 weeks. The course meetings consisted of lectures, pedagogical activities, and in-class laboratory computing experiments. The programming environment used in the course was DrJava, a simplified version of the Java development environment specifically designed for programming beginners. The course instructor had over 20 years of teaching experience in introductory programming courses and co-authored a popular textbook on the design of Java programs (Cohoon & Davidson, 2006). The CS1112 course was designed to recruit and retain students with diverse demographic background into the computing community.

To this end, four pedagogical practices were implemented. First, computers were available and used throughout all class meetings. Students brought in their own laptops or borrowed from the instructor. Second, course materials such as problem examples were selected based on students' preferences. Third, careers in computing were discussed frequently in the class. Fourth, an inclusive class culture was established and extended beyond the classroom by encouraging interactions among students (Cohoon, 2007; Cohoon & Tychonievich, 2011).

Prior to the course sessions on recursion, the students learned fundamental Java concepts, object manipulation, control structures, problem solving strategies, and methods. On the first course session of the 9th week, the instructor introduced recursion using the Fibonacci function, a selection sort problem, the factorial function, the greatest common divider problem, and an ancestor count problem. He used the boxes representation (Goldschlager & A. Lister, 1982) to illustrate how to evaluate recursive functions. At the end of the course session, he assigned the students to evaluate five recursive methods using a worksheet and to formulate a recursive method to draw the Sierpinski carpet (Mandelbrot, 1983). In the second course session of that week, the teaching assistant first discussed the Sierpinski carpet assignment and similar fractal problems. He then introduced three ways of tracing recursive methods: the tree approach (Haynes, 1995; Kruse, 1982), the substitution approach (Abelson, Sussman, & Sussman, 1996), and the block diagram approach (Troy & Early, 1992). On the third course session of that week, the teaching assistant discussed recursive void methods and illustrated their behaviors using methods involving printing statements. Then, the students took a 40-minute quiz on recursion.

Following the quiz, 28 of the 60 students participated in the follow-up interviews. These participants' quiz performance reflected the larger trends in understanding of recursion. This subsample reflected the diverse demographic and academic backgrounds of the full course enrollment (see Table 6 for demographic information for the subsample).

Table 6

Demographics of the whole sample and the subsample of interview participants

Demographics	Subsample(n=28)
Gender	
- Female	17 (61%)
- Male	11 (39%)
Ethnicity	
- American Indian or Alaska Native	0 (0%)
- Asian	6 (21%)
- Black or African American	2 (7%)
- Native Hawaiian or other Pacific Islander	0 (0%)
- Hispanic or Latino	2 (7%)
- White	18 (64%)
Major or intended major	
- Biology	4 (14%)
- Commerce	2 (7%)
- Computing	3 (11%)
- Engineering	6 (21%)
- Mathematics	4 (14%)
- Medical science	1 (4%)
- Psychology	7 (7%)
- Unknown	6 (21%)
Academic year	
- First year	16 (57%)
- Second year	5 (18%)
- Third year	2 (7%)
- Fourth year	3 (11%)
- Unknown	2 (7%)

Study 1

Design

Study 1 tests the three quantitative hypotheses. A quiz including four program evaluation tasks and one program comprehension task was administrated to all participants. Mental models that participants exhibited in their program evaluation responses were categorized using an adapted version of an existing categorization scheme. Mental models that participants exhibited in their program comprehension responses were categorized using a simplified version of the existing categorization scheme. The comprehension-based results was used to adjust the stability of copies model determined using the program evaluation tasks. Statistical analyses were performed to determine the relationship between mental model manifestation and situational constraint imposed by each problem. For further analysis, stability of the copies model was determined using the frequency with which the copies model manifested.

Operational Definitions of Key Variables

Mental Models of Recursion

Mental models of recursion are operationally defined as the patterns of thinking presented in participants' program evaluation traces or their descriptions of the goal of a function. Evaluation traces were coded according to an adapted version of Götschi's (2003) categorization scheme (see Appendix 1 for the original categorization scheme). Götschi's scheme provides detailed coding explanations, and its content validity was verified by experts of introductory CS courses. According to Götschi, raters first code the trace component (i.e., active flow, base case, and passive flow) and then categorize the trace based on the combinations of the three codes (see Appendix 1 for detailed coding scheme).

The specific procedures and codes were adapted to accommodate the theoretical assumptions and the characteristics of participants' traces in the present study (see Appendix 2 for the adapted categorization scheme). In the present study, codes at the trace component level were disregarded because such coding method prematurely presumes that the copies model is psychologically composed in such way. Additionally, not every trace has all three trace components, and several codes across components are not independent from each other. As a result, traces were coded only at the trace level and directly categorized into various mental models.

Several new models emerged from the data. The previously identified active model (Götschi, 2003) was found to have eight identifiable sub-models in the current study. The original one was identical to the sub-model named combine-all-after-base-case in the adapted categorization scheme. Other sub-models varied by how evaluation ends and whether there is an output at each invocation. Other new models emerged from the data included: a bottom-up model, a shortcut model, and a function description model. The existence of these models can be attributed to the instructional and task contexts of the course, including the format of materials, presented exercises, and the quiz problems (see discussion of course observation field notes, below).Situational Constraint

Situational constraint is the extent to which features of a task prevent the computer-as-agent p-prim from operating in inappropriate ways through the mediation of prior knowledge. According to the knowledge-in-pieces theory, activated knowledge elements specify a context for further knowledge activation (diSessa, 1993). An activated computer-as-agent p-prim alone specifies a context that favors problematic schemas. However, perception of specific task features may activate certain knowledge elements and a resulting context favors productive schemas. As a result, the presence of these task features constrains the predominant influence of the computer-as-agent p-prim over further knowledge activation.

In this study, situational constraint is operationally defined using two task dimensions: call structure and parameter type. Each dimension has two levels. Call structure is either prefix or tail. Parameter type is either number or list. As discussed in Chapter 2, it is assumed that the prefix call structure imposes a higher situational constraint than the tail call structure due to its capacity to withhold the operation of the computer-as-agent p-prim. Also, it is assumed that the number parameter has a higher situational constraint than the list parameter due to participants' ability to view numeric operations in their entirety—an ability underdeveloped for lists—constrains the operation of the computer-as-agent p-prim. Thus, the combination of the two dimensions generates four tasks with varying levels of situational constraint. As shown in

Table 7, a method with a prefix call structure and number parameters imposes the highest level of situational constraint. A method with a tail call structure and number parameters, or a prefix call structure and list parameters imposes moderate situational constraints. A method with a tail call and list parameters imposes low situational constraints.

Table 7
Situational constraint of the four program evaluation tasks

		Situational constraint imposed by call structure	
		Prefix	Tail
Situational constraint imposed by parameter type	Number	High	Medium
	List	Medium	Low

Stability of the Copies Model

Mental models are unstable and constantly evolving (Norman, 1983), but they can be described as stable when they establish certain level of entirety, stored in and retrieved from long-term memory as a whole unit (Doyle & Ford, 1998). Thus, the stability of a certain mental model can be measured by the frequency with which it manifests across multiple tasks. In this study, stability of the copies model is operationally defined as the frequency of copies model manifestation among the four program evaluation tasks.

Instruments

Mental Models of Recursion Test

The mental models of recursion test consisted of four program evaluation tasks and one program comprehension task (see Appendix 3). In the program comprehension task, participants described what a given recursive function achieved in plain language. In the program evaluation tasks, participants evaluated recursive functions and showed their evaluation processes in writing. The four recursive functions had different combinations of call structure and parameter type. One had a prefix call structure and number parameters; one had a prefix call structure and a list parameter; one had a tail call structure and number parameters; and one had a tail call structure and a list parameter. The 24 different orders of task presentation were evenly distributed among participants.

The usage of the program evaluation task derives from Kahney's (1983) study on novices' knowledge of recursive functions. Kahney used a tail-recursive function and an embedded recursive function, both of which produced the same outcome. Participants were asked to determine whether the programs could solve a given problem and to explain "how it does it (in your own words), or, if it won't, say why it doesn't (again in your own words)" (p. 236). This task effectively differentiated multiple understandings of recursive function.

However, Kahney's request for explanations in his study resulted in unstructured responses, from which it was difficult to interpret participants' mental models of recursion.

The request for explanation is made much clearer by asking participants to simulate step-by-step how a recursive function is executed. Götschi (2003, p.80), for example, asked participants to "trace through the program with the program call given. Please show as much of your thought processes as possible, either by using diagrams to represent what you understand to be happening or by writing English [plain language] statements." In response, participants described their thought processes in detail, often in the form of a trace. Sometimes, researchers phrase the request in terms of specific programming language. Mirolo (2010, appendix, p.1), for example, required participants to use the substitution method (Abelson, Sussman, & Sussman, 1996) specific to the Scheme programming language: "based on the Scheme's computation model, show the key evaluation steps for the expression. In particular, report all the recursive procedure calls." This kind of request further constrains participants to respond in particular forms. Still, participants' responses showed lack of details and considerable ambiguity. Götschi (2003) reported that only 49% of participants (n=172) clearly simulated program execution, 9% showed how inputs were calculated to generate outputs with no reference to the execution mechanism, 30% responded a mixture of the simulation and the calculation, and 12% only provided the outputs without explaining their thought processes.

Drawing on the lessons learned from these past efforts to capture participants' mental models, the evaluation tasks were phrased as execution simulation tasks. The participants were explicitly asked to show the process through which a given input produced certain output.

However, it was possible that specifying response forms might increase the situational constraint in all tasks, which would diminish the effect of task features (i.e., parameter type and call structure). To compensate for the heightened situational constraint, a program comprehension task was included in the test. This type of task only asked participants to predict the inputs based on the given outputs, thus it imposed a relatively low level of situational constraint.

Observation Field Notes

To inform results interpretation as necessary, observation field notes were taken to capture the instructions the participants received on recursion. The field notes were designed to record contents of the instructions and participants' reactions to the instructions. Particularly, the notes recorded in details the representations that the instructors used to illustrate recursive processes.

Procedure

The researcher started to observe the class from sessions on the concept of method, which laid a foundation for the concept of recursion. Observation continued through three course sessions on recursion until the mental models of recursion test was administrated as a quiz. In addition, the researcher collected relevant textbook chapters, lecture notes, worksheets, and homework assignments.

The course teaching assistant administrated the quiz in pencil-and-paper format at the end of the third class session on recursion. He randomly distributed the quiz sheets which presented the tasks in varied sequences. Time limit was 40 minutes. Participants could leave once they submitted their answer sheets.

Analysis

Categorizing Mental Models of Recursion Exhibited In the Program Evaluation Tasks

Participants' evaluation traces were categorized based on the adapted categorization scheme for mental models of recursion (see Appendix 2). A minority of the traces were not immediately classifiable (none for the number-prefix method, 18.3% for the number-tail method, 20.0% for the list-prefix method, and 11.7% for the list-tail method; see Appendix 4 for uncertain categorization in each model). Fortunately, the copies model traces were all classifiable without any uncertainty. Thus, all the cases were preserved by collapsing all other categories into a non-copies model category. However, there was one trace categorized as the active model that could be alternatively categorized as the copies model. With no interview data available for triangulation, this case was eliminated from hypothesis testing. The resulting sample size was 59.

Testing Hypothesis 1

Hypothesis 1: Participants are more likely to exhibit the copies model when evaluating a recursive function with a prefix call structure than evaluating a recursive function with a tail call structure.

This hypothesis is tested using two one-sided Z tests for dependent samples at the confidence level of 95%. The first test compares proportions of participants who demonstrate the copies model in evaluation of the number-prefix method and in evaluation of the number-tail method. The second test compares proportions of participants who demonstrate the copies model in evaluation of the list-prefix task and in evaluation of the list-tail task.

Testing Hypothesis 2

Hypothesis 2: Participants are more likely to exhibit the copies model when evaluating a recursive function with a number parameter than evaluating a recursive function with a list parameter.

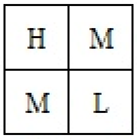
This hypothesis is tested using two one-sided Z tests for dependent samples at the confidence level of 95%. The first test compares proportions of participants who demonstrate the copies model in evaluation of the number-prefix method and in evaluation of the list-prefix method. The second test compares proportions of participants who demonstrate the copies model in evaluation of the number-tail task and in evaluation of the list-tail task.

Testing Hypothesis 3

Hypothesis 3: If participants exhibit the copies model in some but not all task situations that represent four possible combinations of call structure and parameter type, their successful performance will be clustered around tasks with higher situational constraints. Performance patterns representing such trend should occur more frequently than those contradicting it.

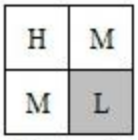
Participants are classified based on their performance pattern. There are 16 possible combinations of the copies model manifestation among the four tasks. These combinations are classified as a categorical variable called *performance pattern*.

The copies model may show in all four tasks:

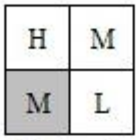


Pattern 1

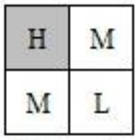
The copies model may show in three of the four tasks:



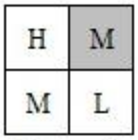
Pattern 2



Pattern 3

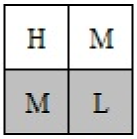


Pattern 4

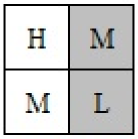


Pattern 5

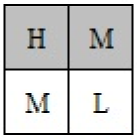
The copies model may show in two of the four tasks:



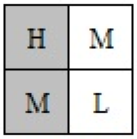
Pattern 6



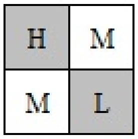
Pattern 7



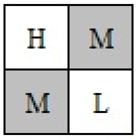
Pattern 8



Pattern 9

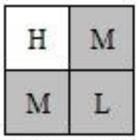


Pattern 10

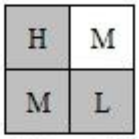


Pattern 11

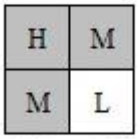
The copies model may show in one of the four tasks:



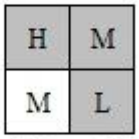
Pattern 12



Pattern 13

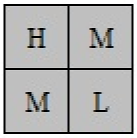


Pattern 14



Pattern 15

The copies model may show in none of the four tasks:



Pattern 16

The 16 performance patterns may be consistent, inconsistent, or uninformative to Hypothesis 3. When participants demonstrate the copies model in tasks with higher situational constraint but not in those with lower situational constraint (i.e., pattern 2, 6, 7, and 12), the clustering patterns are consistent with Hypothesis 3. When participants demonstrate the copies model in tasks with lower situational constraint but not in those with higher situational constraint (i.e., pattern 3, 4, 5, 8, 9, 10, 11, 13, 14, and 15), the clustering patterns are inconsistent with Hypothesis 3. When participants demonstrate the copies model in all tasks (pattern 1) or in none of the problems (combination 16), no informative clustering pattern is available to evaluate Hypothesis 3.

The participants who demonstrate the two hypothesis-uninformative patterns are not included in this analysis, thus there remain 14 patterns. The aggregated expected proportions are 28.6% (4 out of 14) for the hypothesis-consistent patterns and 71.4% (10 out of 14) for the hypothesis-inconsistent patterns. An exact test of goodness-of-fit determines whether the observed distribution differs significantly from a random distribution. Further, one-sided Z-tests are used to determine whether the observed proportions are different from their respective expected proportions in the expected directions. Specifically, the aggregated proportion of the hypothesis-consistent patterns is predicted to be higher than expected, while the aggregated proportion of the hypothesis-inconsistent patterns is predicted to be lower than expected. To control for Type I error using the Bonferroni correction method, the overall alpha level (.05) is divided up by the total number of tests (2), making a critical alpha value of .025 for each individual test.

Categorizing Mental Models of Recursion Exhibited In the Program Comprehension Task

Participants' answers to the program comprehension question are categorized based on the simplified categorization scheme for mental models of recursion. Their answers are coded as the potential-copies model if they correctly complete the comprehension task. If they give incorrect or partial answers, their answers are coded as the non-copies model.

Determining Stability of the Copies Model

For the purpose of subsequent analyses, participants are also classified according to the stability of their demonstrated copies model. As shown in Table 8, stability of the copies model is ranked by the proportion of the five tasks in which model manifested. The potential-copies model in the program comprehension task is counted as a copies model. The resulting stability of the copies model has six levels: high, moderate high, moderate, moderate low, low, and absence.

Table 8
Stability of the copies model ranked by the proportion of the five tasks in which model manifested

Manifestation of the copies model	Stability of the copies model
In all five tasks	High
In four tasks	Moderate high
In three tasks	Moderate
In two tasks	Moderate low
In one task	Low
In none of the tasks	Absence

Study 2

Design

Study 2 attempts to validate the three qualitative suppositions regarding the operation of the computer-as-agent p-prim in evaluation of recursive function. On the basis of their test performance, a demographically diverse sub-sample reflecting the larger trends in understanding were selected to participate in additional, in-depth clinical interviews. As described below, interview transcripts are qualitatively analyzed to distill patterns of interpretations indicating the structure, function, and coordination modes of the computer-as-agent p-prim. The coding scheme for various coordination modes of the computer-as-agent p-prim was developed through a deliberate, inductive procedure and utilized to analyze interview transcripts.

Operational Definitions of Key Concepts

The computer-as-agent p-prim is operationally defined as a general perception-action pattern: individuals perceive program codes as actions the computer will take and then seek results for the actions using programming knowledge. Each action must be matched with a result.

Programming schemas are operationally defined as specific perception-action patterns: individuals perceive program codes as specific operations and then implement the operations. Programming schemas can be productive or unproductive depending whether they accurately represent the official definitions of the program codes.

Coordination is operationally defined as the situation that the computer-as-agent p-prim and productive programming schemas are simultaneously activated and then simultaneously completed. Tentatively, there can be three modes of coordination: incoordination, conditional coordination, unconditional coordination. Incoordination refers to the mode that coordination does not occur in any of the tasks. Conditional coordination refers to the mode that coordination occurs only in tasks with certain features. Unconditional coordination refers to the mode that coordination occurs consistently across all the tasks.

Procedure

As shown in Table 9, half of the class (32 out of 60 participants) expressed interest in participating in the follow-up interviews. To ensure an adequate sample size, all the volunteers were invited. Of these, 28 participants participated in the interview sessions. An exact test of goodness-of-fit indicated no statistically significant difference between the distribution of the copies model stability in this subsample and that in the whole sample (p = .982). The subsample was representative of the whole sample in quiz performance.

Table 9
Distribution of adjusted stability of the copies model among the interview participants and all participants

Adjusted stability of the copies model	Interview participants Frequency (proportion)	All participants Frequency (proportion)
Absence	6 (21%)	11 (19%)
Low	9 (32%)	18 (31%)
Moderate low	4 (14%)	11 (19%)
Moderate	3 (11%)	8 (14%)
Moderate high	3 (11%)	5 (8%)
High	3 (11%)	6 (10%)
Total number of participants	28	59

Each interview participant met with an interviewer in a private meeting room for 60 minutes. The interviewer reviewed the purpose of the study, the procedure of the interview, and the consent forms. Once the participant signed the consent forms, the interviewer started the audio recording.

Clinical Interview

Clinical interview is "a flexible method of questioning intended to explore the richness of children's thought, to capture its fundamental activities, and to establish the child's cognitive competence" (Ginsburg, 1981, p.4). In a clinical interview, the interviewer typically uses open-ended questions or tasks to initiate the conversation with the participant. For example, the interviewer may describe a phenomenon and then ask the participant to make predictions or explanations about it or to think aloud while performing a task. As the conversation unfolds, the interviewer makes "improvisational moves" to encourage more elaboration, clarify or triangulate the participant's meanings, or maintain a comfortable and focused conversational environment (Lee, Russ & Sherin, 2008, p.1724).

Compared to other methods commonly used in studies of conceptual change, clinical interview method has unique advantages. For instance, researchers may gather rich, contextualized data which cannot be gathered in a conventional psychological testing. Also, the clinical interview method is more efficient and controllable than traditional naturalistic observation methods, because it allows for targeted explorations without potentially problematic decontextualization (Ginsburg, 1981).

However, clinical interview has been criticized by socio-cultural theorists for collecting social artifacts rather than knowledge manifestation (Bannon & Bødker, 1991). Some radical criticism fundamentally questions the existence of knowledge as a knowable entity (e.g., Roth, 2008). Some mild criticism questions the reliance on the subjects' responses to infer their knowledge without examining the interactional patterns around the responses (Halldén, Haglund, & Strömdahl, 2007). In response to these criticisms, clinical interview practitioners, while maintaining that knowledge is ontologically valid (e.g., Vosniadou, 2007), started to explore the ecological validity of clinical interview (diSessa,

2007) and take interaction analysis into account when conducting knowledge analysis (e.g., Sherin, Krakowski, & Lee, 2012). In a recent educational research symposium, aspects of this integrated account of clinical interview methodology were explicitly discussed and evaluated (e.g., interviewer's revoicing strategy, (diSessa, Greeno, & Michaels, 2012; interactional features surrounding knowledge manifestation, Brown et al., 2012). Although a promising and interesting direction, these integrated analyses necessarily double the workload of data analysis and require additional assumptions to infer participants' knowledge. Instead of implementing a full-bloomed integrated knowledge-interaction analysis, this study was designed to prevent potential interactional effects at the data collection stage and then follow up with a lite inspection at the data analysis stage.

In the current study, I employed two strategies to minimize the unwanted influences of social interaction on knowledge manifestation in clinical interviews. First, the clinical interviews were grounded in naturalist data. In a typical clinical interview, the interviewer asks the participants questions without references to their performance in other contexts, and the participants have to improvise answers (Roth, 2008). Such a setting potentially allows the interviewing social dynamic to exert unchecked influence on knowledge manifestation (Halldén, Haglund, & Strömdahl, 2007). In order to minimize this unwanted influence, the design of this study bundled and sequenced the collection of artifacts (Given, 2008), the cued retrospective reporting (van Gog, Paas, van Merriënboer, & Witte, 2005), and the probes (Klein & Calderwood, 1996) techniques to target the same knowledge. Participants' quiz responses collected in Study 1 were the artifacts of their knowledge manifestation in a naturalistic setting. These artifacts helped cuing participants' recollection during retrospective reporting in the clinical interviews. The interviewer only asked probe questions after participants complete their reports in order to minimize the influences of interaction over the retrospective reports.

The second strategy was to frame clinical interview as a derivative of inquiry activities that naturally occur in everyday life (diSessa, 2007). Although it was impossible to completely avoid an asymmetrical relationship between the interviewer and the participant, multiple strategies were used to psychologically prepare the participants for natural and genuine inquiries. First, the interviewer created a non-authoritative, casual conversational environment by intentionally dressing informally and selecting a room with casual decor. At the beginning of the interview, the interviewer explicitly stated the purpose of the interview as a method of scientific investigation rather than a performance assessment. During the interview, the interviewer maintained the established conversational environment using rhetorical techniques. For instance, the interviewer used discourse markers such as "so the question is…" to connect a foregoing problem description and the question to convey a sense of natural inquiry. Also, the interviewer refrained from giving evaluative or corrective messages to the participant throughout the interview.

Having the appropriate interactional pattern established, the interviewer asked the participants to describe their thought processes during their attempts to solve the problems presented on the quiz. The interviewer demonstrated how to retrospectively report one's own thought processes using a simple mathematical problem. After briefly reviewing their own traces, participants reported their thought processes when they attempted the quiz questions. After participants reported on each task, the interviewer asked probe questions to elicit more detailed, in-depth information about the participants' mental processes. The probes targeted the previously identified elements that constitute a contextual configuration, as listed below:

At this moment, which elements were you focusing on?

What information did you use in making this decision?

Were you reminded of any previous experience? What was it?

What was your goal at this moment?

Have you considered the option that ……?

At the end, the interviewer expressed appreciation and provided a $10 compensation to the participants.

Analysis

Screening for Intentional Patterns

All of the interview audio-recordings were transcribed. The transcripts were first screened using the intentional analysis technique (Halldén, Haglund, & Strömdahl, 2007) to eliminate instances of interactions skewed by the social dynamics of the interviews. In intentional analysis, a participant's utterances are viewed as verbal actions performed to reach certain goals. One source for verbal action is the participant's beliefs about the subject matter irrespective of any particular situation. The other source is the participant's interpretation of the particular interview situation in terms of duties, norms, and opportunities (von Wright, 1971). The participant's interpretations of the interview situation may distort her or his knowledge construction and presentation of the subject. For example, when participants perceive the interviews as a test situation in which questions are intended to make them error-prone, they would dodge the questions and stick to the knowledge that they are confident about (Halldén, Haglund, & Strömdahl, 2007). Interview segments with occurrence of such distortion would be eliminated from the subsequent analysis.

Overall, the participants' intentions in their verbal actions were compatible with the interviewer's expectations. They honestly and plainly reported in details their thought processes during the quiz. There were a few instances where participants misunderstood the interviewer's questions or requests, but these misunderstandings were quickly corrected during the interviews. A few participants appeared to be nervous, embarrassed, or defensive at the beginning, but as the interviewer constantly reinforced a friendly and safe atmosphere, they gradually became relaxed and comfortable with the conversation. Therefore, these participants' transcripts were still effective after eliminating the segments involving misunderstandings and negative emotions. As a result, all 28 cases were preserved for the analyses described below.

Data Segmentation

Each transcript was segmented into four sections: 1) *retrospective report of thought processes*, during which participants retrospectively reported their thought processes on the quiz questions, and then the interviewer asked questions to clarify participants' reports (the sequence of reporting the five tasks varied depending on the order with which the participants completed the tasks on the quiz day); 2) *answer review and probing*, during which participants reviewed the correct answers and compared them to their own answers, and then the interviewer asked probe questions to gain further information about participants' understanding of recursive function; 3) *tutoring*, during which the interviewer tutored the participants on recursion if they demonstrated poor understanding and were willing to learn; 4) *evaluation of instructions*, during which the interviewer asked the participants to evaluate the effectiveness of instructions on recursion, particularly the different types of activities, representations, and assignments. For the purpose of this study, only the first two sections were used for analysis.

Participants' reports for each task were segmented into a series of mental transaction units. As shown in Figure 8, three domains comprise a mental transaction unit: contextual configuration, three levels of information processing, and action. Contextual configuration is composed of the information the participants report attending to in the environment and the information they hold in their short-term memory at the moment. Three levels of information processing are the sensory level, the p-prim level, and the symbolic schema level (diSessa, 1993). The contextual configuration shaped by previous transactions supplies the sensory input, which in turn activates and instantiates the computer-as-agent p-prim. The instantiated computer-as-agent p-prim then activates and coordinates with relevant symbolic schemas. The active p-prim and schemas generate internal actions (e.g., storing information in short-term memory) and external actions (e.g., writing a note on quiz sheet). These actions reshape the contextual configuration, which will serve as the starting point for the next mental transaction unit.

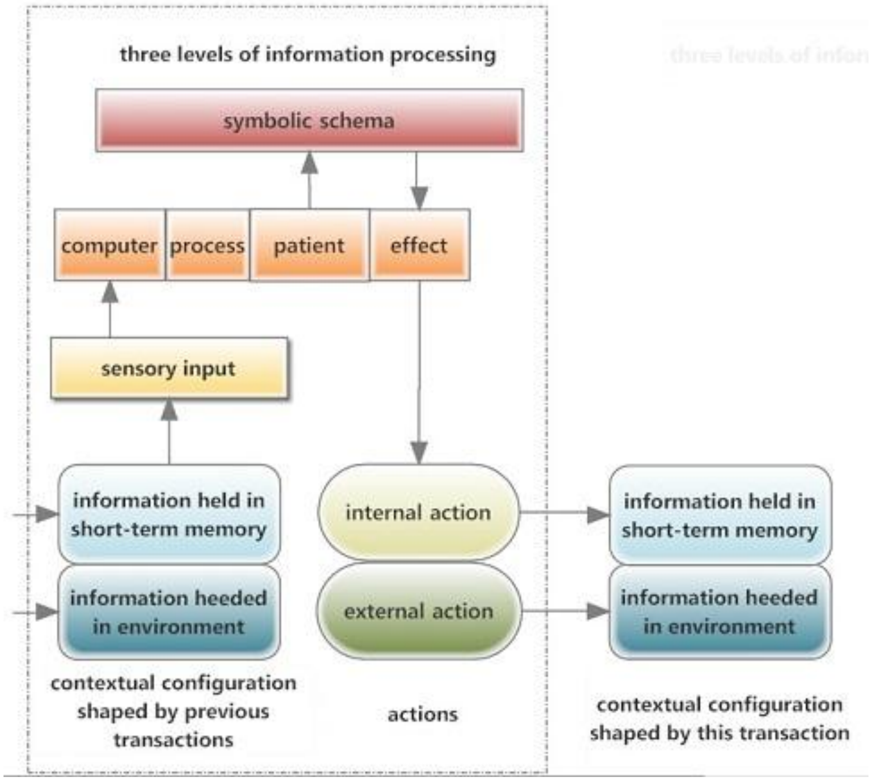

three levels of information processing
symbolic schema
computer
process
patient
effect
sensory input
information held in short-term memory
information heeded in environment
internal action
external action
information held in short-term memory
information heeded in environment
contextual configuration shaped by previous transactions
actions
contextual configuration shaped by this transaction

Figure 8. Mental transaction unit

For example, during evaluation of a recursive method, a participant encountered the statement “int n = list.size ();”, which initializes an integer n with the number of elements in a given list. In the interview, the participant reported how she evaluated this statement:

> “It says n equals list size, so I looked at the list that was given in the problem, and there are four numbers in it, so I got the size of the list which is 4, so I set n equal to 4.”

At the beginning of this unit, the participant focused on the statement “int n = list.size ();”. This sensory input activated and instantiated the computer-as-agent p-prim, which in turn activated the symbolic schema of this particular statement. Operation of the schema generated a result which filled the effect slot of the instantiated computer-as-agent p-prim. The participant acted accordingly, writing down a note “n = 4” on her quiz sheet.

Selection of Mental Transaction Units

The contextual configurations of the following elements satisfy the activation requirement for the computer-as-agent p-prim, thus they were selected for analysis:

1. Original invocation specified in the problem statement (e.g., the problem statement "what output does invocation g (10, 3) produce?" along the code of method g);
2. Non-recursive statement, or a statement not involving a recursive call (e.g., the statement "Boolean b = (u <= v);" in the method q").
3. Recursive statement, or a statement involving a recursive call (e.g., the statement "result = m * p (m, n - 1);" in the method p);

Sub-invocation (e.g., g (10, 2) written by the participants to denote a new method invocation);Table 10 shows distributions of each element category among the four tasks.

Table 10
Distributions of each type of contextual configurations among the four tasks

Mental transaction units	Tasks			
	Number-prefix	Number-tail	List-prefix	List-tail
Original invocation	1	1	1	1
Sub-invocation	4	3	4	3
Recursive statement	4	3	4	3
Non-recursive statement	15	11	25	20
Total	20	15	30	24

Identifying the Computer-as-agent P-Prim for Validation of Suppositions 1, 2, and 3

Based on the principles for identifying p-prims (diSessa, 1993), interview transcripts were qualitatively analyzed to distill patterns of interpretation which indicated the structure, function, and coordination modes of the computer-as-agent p-prim. The principles can be roughly organized into several categories that respectively focus on behavioral attributes, developmental attributes, class properties of p-prims, and analytical strategies. Due to the scope of this study, only behavioral attributes and analytical strategies are reviewed here.

According to the principles on behavioral attributes, it is generally productive to examine the “misconception” cases to identify p-prims (“principle of discrepancy”; diSessa, 1993, p.125). However, it is worth noting that p-prims operate across a breadth of cases regardless of whether a response is correct or incorrect evaluated by scientific norms (“principle of coverage”; p.121). Good opportunities to identify p-prims occur when participants feel that a given situation does not need an explanation (“principle of obviousness”; p.121), or they claim an explanation with great satisfaction (“principle of impenetrability”; p.121). At such moments, the particular words and phrases that participants use often signal the underlying p-prims (“principle of strong vocabulary”; p.122), especially when their vocabulary appears to be readymade for representing the given situation (“principle of ready availability”; p.123).

In order to characterize the p-prims, it is particularly useful to consider what they enable the participants to do in their interactions with the world (“principle of functionality”; p.123). Usually, participants’ responses change during a complete attempt (“principle of dynamics”; p.123). Their first responses signal those p-prims of high cuing priority in that situation. As they attend to more features of the situation, other p-prims or schemas may be brought forth. If certain ideas persist through or finalize the responses, they may indicate the underlying p-prims of high reliability priority in the situation.

If a p-prim is characterized correctly, it should manifest itself consistently (“principle of invariance"; diSessa, 1993, p.124) in multiple applicable situations ("principle of diverse evidence"; p.124). If any evidence does not agree with the principle of invariance, the p-prim's description must be revised to achieve consistency ("principle of redescription"; p.124). This triangulation does not need to be confined within a single dataset. Data from other sources, as long as qualified for such analysis, can be used to evaluate the working hypothesis ("principle of scavenging data"; p.124).

During the process of identification, as qualified cases accumulated, analysis was gradually evolving to inform the extent to which suppositions 1, 2 and 3 were supported:

Supposition 1: Interpretations generated by the computer-as-agent p-prim are characteristic of a sense of intuitive obviousness and satisfaction.

Supposition 2: Interpretations generated by the computer-as-agent p-prim serve participants well in many cases rather than causing problems all the time.

Supposition 3: Interpretations generated by the computer-as-agent p-prim are sensitive to local context, thus they are likely to change in response to change in local context.

Coding for Schemas Relevant to Construction of Mental Models of Recursion

The selected mental transaction units were further coded to identify programming schemas that were relevant to construction of mental models of recursion. Initially, couplings of perceptions and actions were openly coded. These initial codes were condensed and then applied to the data again. This procedure was repeated for several times until resulting codes were comprehensive and parsimonious. The identified schemas were further categorized as productive or unproductive based on whether their operations result in the copies or non-copies model of recursive function.

Classifying Participants on the basis of Coordination Modes

Participants are classified on the basis of how their computer-as-agent p-prim coordinates with the productive schemas. When the computer-as-agent p-prim and productive schemas co-occur among all tasks, the participants are classified as operating in an unconditional coordination mode. When the computer-as-agent p-prim and productive schemas co-occur only in tasks with certain features, participants are classified as operating in a conditional coordination mode. When the computer-as-agent p-prim and productive schemas do not co-occur in any of the tasks, participants are classified as operating in an incoordination mode. The unconditional coordination mode and conditional coordination mode include subcategories differentiating stability of coordination. If coordination persists through the evaluation process, the mode is labeled as stable. Otherwise, the mode is labeled as unstable.

Integrated Analyses

Triangulation and Retesting Hypotheses

The interview transcripts available for the 28 interview participants were used to triangulate their traces obtained through analysis in Study 1. Omitted steps in the traces were filled with contents available in the clinical interview transcripts obtained in Study 2. Shortcut traces or function descriptions were expanded to through triangulation using the participants' elaborations in the interviews. Using the augmented traces, the data were recoded using the adapted categorization scheme for mental models of recursion.

The triangulated trace data represented a more detailed observation of the participants' knowledge systems than the untriangulated trace data, resulting in the reclassification of several traces. However, when the interviewer asked the participants to complete the tasks with more stringent requirements (e.g., no shortcut or abbreviation allowed) she imposed a higher situational constraint than that imposed by the quiz. Thus, triangulated trace data complement rather than replace the original Study 1 trace data.

Retesting the three hypotheses using the triangulated trace data provides two sets of results (untriangulated and triangulated), which are interpreted jointly to hedge the risk of committing Type I and Type II errors. Untriangulated trace data are prone to Type II error, because they preclude the interpretation of aspects of participants' mental processes that do not have corresponding written aspects. Specifically, the hypotheses are likely to be inappropriately unsupported, because many traces categorized as non-copies in the higher-constraint tasks can be potentially categorized as an instantiation of the copies model when supplemented by an articulation of the underlying reasoning processes. Interview data indicates that some of these participants were indeed capable of demonstrating the copies model in the higher-constraint tasks but not in the lower-constraint tasks. Further, there could be additional cases among the participants who did not participate in the interviews.

Conversely, results from the triangulated trace data are prone to Type I error, because the interview context imposed a higher level of constraint on mental model construction than the quiz context did. Thus the additional copies models associated with the higher-constraint tasks could be artifacts of the interview context and not representative of the participants' performance during the quiz as taken in the classroom context. Therefore, tests using only these data might inappropriately bias results to support the hypotheses.

Validation of Assumptions Regarding Situational Constraints

Clinical interview data obtained in Study 2 are compared to the assumed imposition of situational constraints to validate the manipulation of independent variables. Specifically, descriptions of mental processes are compared against the assumption that the prefix call structure and the number parameter would prevent the computer-as-agent p-prim from operating in inappropriate ways.

Substantiating Supposition 4

The stability of the copies model determined in Study 1 and the coordination mode determined in Study 2 are taken together to evaluate the fourth supposition:

Supposition 4: Stability of the copies model is associated with coordination mode. The more coordinated the operation of the p-prim, the more stable the copies model will be.

A cross-tabulation presents the trend of co-occurrence between the level of stability of the copies model and coordination mode. The statistical significance of this association is determined using the Fisher's exact test at the confidence level of 95% and the Spearman's Rank Order correlation at the confidence level of 95%.

CHAPTER 4

FINDINGS

The results of the current study are presented in three sections: Study 1, Study 2 and Integrated Analyses. In the results of Study 1, participants' program evaluation performance data (triangulated and untriangulated trace data) are categorized using the adapted categorization scheme for mental models of recursion and then analyzed to test the three hypotheses. Both sets of results are presented to assess convergence and divergence pertinent to the hypotheses tested. In addition, the stability of participants' copies models is determined using the untriangulated trace and program comprehension performance data. In the results of Study 2, analyses of the interview data are presented to substantiate suppositions 1-3. Also presented are programming schemas identified to be relevant to the construction of mental models of recursion. Then, three modes of coordination are determined and tallied based on the patterns of co-occurrence between the computer-as-agent p-prim and identified programming schemas. In the results of the Integrated Analyses, assumptions underlying the three hypotheses are validated using the interview data. Finally, supposition 4 is evaluated using the stability of the copies model and the distribution of coordination modes.

Study 1

Evaluation-based Model Categorization

Participants' evaluation traces were categorized using the adapted categorization scheme for mental models of recursion (see Chapter 3). The report below presents results of the untriangulated trace data followed by results of the triangulated trace data.

Untriangulated Trace Data

As shown in Table 11, 50% of the participants (30 out of 60) demonstrated the copies model in the number-prefix task. For the other three tasks, ~25% (14, 16, and 13 out of 60) of the participants demonstrated the copies model.

Approximately 40% of the participants (24 and 26 out of 60) demonstrated the active model in the number-prefix task and the list-prefix task, as about 60% (34 and 38 out of 60) did in the number-tail task and the list-tail task.

The active model has a range of variants (see Chapter 3 for descriptions of them). The combine-all-after-base-case variant predominately showed in evaluation of the two prefix methods (see Table 11). The revert-to-first and revert-to-second-last variants only showed in evaluation of the two tail methods. The output-from-all and base-case-result variants spread out unevenly among the four tasks. Other variants appeared sporadically.

A few participants demonstrated the bottom-up model in evaluation of the two number methods. A handful of participants (8 out of 60) showed the shortcut model, which only occurred in evaluation of the list-prefix method. A few participants provided function descriptions of the list-prefix method. Lastly, considerable numbers of participants (range 3 to

11 out of 60) demonstrated the step model, and the number-tail method appeared to be most inducive of the step model.

Table 11

Distribution of mental models of recursion among the four program evaluation tasks (untriangulated trace data)

Mental models of recursion	Tasks			
	Number-prefix	Number-tail	List-prefix	List-tail
Copies	30	14	16	13
Active	24	34	26	38
Combine all after base case	18	2	10	
Combine all along the way	2			
Combine last two	1		1	
Revert to first		5		14
Revert to second last		4		5
Output from all	2	12	7	13
Output from all but first		3		
Base case result	1	8	8	6
Bottom Up	3	1		
Shortcut			8	
Function description			3	
Step	3	11	7	9
TOTAL	60	60	60	60
% Triangulation needed	0.0%	18.3%	20.0%	11.7%

Triangulated Trace Data

As shown in Table 12, triangulation using the interview data substantially reduced the percentages of uncertain categorization from an average of 12.5% (range 0.0% to 20.0%) to an average of 8.3% (range 0.0% to 13.3%). The overall performance pattern did not substantially change.

Table 12

Distribution of mental models of recursion among the four program evaluation tasks (triangulated trace data)

Mental models of recursion	Tasks			
	Number-prefix	Number-tail	List-prefix	List-tail
Copies	32	14	22	13
Active	21	34	26	38
Combine all after base case	15	1	8	
Combine all along the way	2			
Combine last two	1		1	
Revert to first		7		14
Revert to second last		4		6
Output from all	1	12	6	12
Output from all but first		2		
Base case result	2	8	11	6
Bottom Up	4	1		
Shortcut			4	
Function description			2	
Step	3	11	6	9
TOTAL	60	60	60	60
% Triangulation needed	0.0%	13.3%	13.3%	6.7%

As shown in Table 13, the interview data confirmed a large majority (64.3%, 82.1%, 89.3%, and 92.9%) of the categorization using the trace data alone. There were 20 instances of recategorization. Half of them involved the copies model. Nine instances were recategorized from non-copies to copies, and 1 instance was recategorized from copies to non-copies.

For the number-prefix method, there were 3 active (combine all after base case) models recategorized to the copies model. These 3 participants abbreviated the passive flow of recursion into mathematical calculation. When prompted to explain how the computer would carry out the calculation, they clearly demonstrated the copies model in their explanations (e.g., participant No.53 explained that: “it would first multiply 1 times 5, and then it would multiply the product of this [1*5=5] times this [5], and then the product of this [5*5=25] times this [5], and the product of this [25*2=125] times this [5]. Go backwards is the order.”). For the same method, there was 1 copies model recategorized to the bottom-up model. This participant’s trace appeared to be a copies model. However, she reported an evaluation strategy from the base case up to the invocation given in the problem statement.

For the list-prefix method, there were two active (combine all after base case) models recategorized to the copies model. Similar to the active-to-copies recategorization explained above, these participants demonstrated the copies model when prompted to explain the passive flow of recursion during the interviews. For the same method, there were 4 shortcut models recategorized to the copies model. These 4 participants took a shortcut to evaluate the list-prefix method, which allowed them to bypass the need to use the copies model. When prompted to extend the evaluation to a complete trace during the interview, these participants were able to generate traces of the copies model without apparent difficulty.

Table 13

Mental models of recursion confirmed or recategorized using the interview data

Task	Confirmed	Recategorized
Number-prefix	23 (82.1%)	3 Active (combine all after base case) → Copies 1 Copies → Bottom up 1, Active (output from all) → Active (base case result)
Number-tail	25 (89.3%)	1 Active (output from all but first) → Active (output from all) 1 Active (combine all after base case) →Active (revert to first) 1 Active (output from all) → Active (revert to first)
List-prefix	18 (64.3%)	2 Active (combine all after base case) → Copies 4 Shortcut → Copies 1 Active (output from all) → Active (base case result) 1 Active (combine all after base case) → Active (base case result) 1 Step → Active (base case result) 1 Function Description → Active (combine all after base case)
List-tail	26 (92.9%)	1 Active (revert to first) → Active (revert to second last) 1 Active (output from all) → Active (revert to first)
Total	92 (82.1%)	20 (17.9%)

Testing Hypothesis 1: Prefix versus Tail

Hypothesis 1: Participants are more likely to exhibit the copies model when evaluating a recursive function with a prefix call structure than evaluating a recursive function with a tail call structure.

This hypothesis was tested with two statistical tests, one comparing proportions of the copies model demonstrated in the evaluation of the number-prefix method and the number-tail method, and the other between the list-prefix method and the list-tail method.

Test 1: Number-Prefix versus Number-Tail

As shown in Table 14, using the untriangulated trace data, 30 participants (50.8% of total 59) demonstrated the copies model in evaluation of the number-prefix method, while 14 participants (23.7% of total 59) did so in evaluation of the number-tail method. A one-sided Z test for dependent samples indicated a statistically significant difference (z = 3.23, 1-tailed p < .001), indicating that the proportion of participants who demonstrated the copies model in high-constraint task was higher than that of participants who did so in the medium-constraint task. When the test was repeated with the triangulated trace data, the results changed little. Thirty two participants (54.2% of total 59) demonstrated the copies model in evaluation of the number-prefix method, and 14 participants (23.7% of total 59) did so in evaluation of the number-tail method. The Z test again indicated a statistically significant difference (z = 3.64, 1-tailed p < .001).

Table 14

Copies model manifested in the number-prefix task and the number-tail task

Situational Constraint	Task	Frequency	Percentage	Z statistics	*p*-values (1-tailed)
		Untriangulated trace data			
High	Number-prefix	30	50.8%	3.23	<.001**
Medium	Number-tail	14	23.7%		
		Triangulated trace data			
High	Number-prefix	32	54.2%	3.64	<.001**
Medium	Number-tail	14	23.7%		

*p < .05, **p < .001

Test 2: List-Prefix versus List-Tail

As shown in Table 15, when the untriangulated trace data were used, 16 participants (27.1% of total 59) demonstrated the copies model in evaluation of the list-prefix method, while 13 participants (22.0% of total 59) did so in evaluation of the list-tail method. A one-sided Z test for dependent samples was performed to determine whether the proportion of participants who demonstrated the copies model in the medium-constraint task was significantly higher than that of participants who did in the low-constraint task. The test failed to indicate a statistically significant difference (z = .73, 1 tailed p = .232 > .05).

When the triangulated trace data were used, 22 participants (37.3% of total 59) demonstrated the copies model in evaluation of the list-prefix method, and 13 participants (22.0% of total 59) did so in evaluation of the list-tail method, yielding a statistically significant difference (z = 1.96, 1 tailed p = .025 < .05).

Table 15

Copies model manifested in the list-prefix task and the list-tail task

Situational Constraint	Task	Frequency	Percentage	Z statistics	*p*-values (1-tailed)
		Untriangulated trace data			
Medium	List-prefix	16	27.1%	.73	.232
Low	List-tail	13	22.0%		
		Triangulated trace data			
Medium	List-prefix	22	37.3%	1.96	.025*
Low	List-tail	13	22.0%		

**p* < .05, ***p* < .001

Testing Hypothesis 2: Number versus List

Hypothesis 2: Participants are more likely to exhibit the copies model when evaluating a recursive function with a number parameter than evaluating a recursive function with a list parameter.

This hypothesis was tested with two statistical tests, one comparing proportions of the copies model demonstrated in evaluation of the number-prefix method and the list-prefix method, and the other between the number-tail method and the list-tail method.

Test 1: Number-Prefix versus List-Prefix

As shown in Table 16, when untriangulated trace data were used, 30 participants (50.8% of total 59) demonstrated the copies model in evaluation of the number-prefix method, and 16 participants (27.1% of total 59) did so in the evaluation of the list-prefix method. A one-sided Z test for dependent samples indicated that the proportion of participants who exhibited the copies model in the high-constraint task was significantly higher than that of participants who did in the medium-constraint task (z = 3.32, 1-tailed p < .001).

When the triangulated trace data were used, the results changed most on the list-prefix method. Thirty two participants (54.2% of total 59) demonstrated the copies model in evaluation of the number-prefix method, and 22 participants (37.3% of total 59) did so in evaluation of the list-prefix method. The Z test also indicated a statistically significant difference (z = 2.74, 1-tailed p = .003 < .05).

Table 16

Copies model manifested in the number-prefix task and the list-prefix task

Situational Constraint	Task	Frequency	Percentage	Z statistics	*p*-values (1-tailed)
		Untriangulated trace data			
High	Number-prefix	30	50.8%	3.32	<.001**
Medium	List-prefix	16	27.1%		
		Triangulated trace data			
High	Number-prefix	32	54.2%	2.74	.003*
Medium	List-prefix	22	37.3%		

**p* < .05, ***p* < .001

Test 2: Number-Tail versus List-Tail

As shown in Table 17, when untriangulated trace data were used, 14 participants (23.7% of total 59) demonstrated the copies model in evaluation of the number-tail method, and 13 participants (22.0% of total 59) did so in evaluation of the list-tail method. A one-sided Z test for dependent samples failed to indicated that the proportion of participants who demonstrated the copies model in the medium-constraint task was higher than that of participants who did in the low-constraint task (z = .95, 1-tailed p = .172 > .05). Triangulated trace data did not change this result.

Table 17

Copies model manifested in the number-tail task and the list-tail task

Situational Constraint	Task	Frequency	Percentage	Z statistics	*p*-values (1-tailed)
Untriangulated trace data					
Medium	Number-tail	14	23.7%	.95	.172
Low	List-tail	13	22.0%		
Triangulated trace data					
Medium	Number-tail	14	23.7%	.95	.172
Low	List-tail	13	22.0%		

*$p < .05$, **$p < .001$

Performance Pattern

Participants were further classified based on their performance patterns. There are 16 possible combinations of the copies model manifestation among the four program evaluation tasks. Table 18 and Table 19 show the distributions of the untriangulated and triangulated trace data, respectively.

Table 18

Distribution of performance patterns among participants (untriangulated trace data)

	Performance Pattern		Frequency (proportion)		Performance pattern		Frequency (proportion)
1	H_{n-p}	M_{n-t}	6 (10.2%)	9	▨ H_{n-p}	M_{n-t}	2 (3.4%)
	M_{l-p}	L_{l-t}			▨ M_{l-p}	L_{l-t}	
2	H_{n-p}	M_{n-t}	0	10	▨ H_{n-p}	M_{n-t}	0
	M_{l-p}	▨ L_{l-t}			M_{l-p}	▨ L_{l-t}	
3	H_{n-p}	M_{n-t}	4 (6.8%)	11	H_{n-p}	▨ M_{n-t}	0
	▨ M_{l-p}	L_{l-t}			▨ M_{l-p}	L_{l-t}	
4	▨ H_{n-p}	M_{n-t}	1 (1.7%)	12	H_{n-p}	▨ M_{n-t}	10 (16.9%)
	M_{l-p}	L_{l-t}			▨ M_{l-p}	▨ L_{l-t}	
5	H_{n-p}	▨ M_{n-t}	0	13	▨ H_{n-p}	M_{n-t}	0
	M_{l-p}	L_{l-t}			▨ M_{l-p}	▨ L_{l-t}	
6	H_{n-p}	M_{n-t}	1 (1.7%)	14	▨ H_{n-p}	▨ M_{n-t}	0
	▨ M_{l-p}	▨ L_{l-t}			▨ M_{l-p}	L_{l-t}	
7	H_{n-p}	▨ M_{n-t}	9 (15.3%)	15	▨ H_{n-p}	▨ M_{n-t}	0
	M_{l-p}	▨ L_{l-t}			M_{l-p}	▨ L_{l-t}	
8	▨ H_{n-p}	▨ M_{n-t}	0	16	▨ H_{n-p}	▨ M_{n-t}	26 (44.1%)
	M_{l-p}	L_{l-t}			▨ M_{l-p}	▨ L_{l-t}	

Note. H_{n-p} denotes the number-prefix method which has high situational constraint. M_{l-p} denotes the list-prefix method which has medium situational constraint. M_{n-t} denotes the number-tail method which has medium situational constraint. L_{l-t} denotes the list-tail method which has medium situational constraint.

Table 19

Distribution of performance patterns among participants (triangulated trace data)

	Performance pattern		Frequency (proportion)		Performance pattern		Frequency (proportion)
1	H_{n-p}	M_{n-t}	8 (13.6%)	9	▨ H_{n-p}	M_{n-t}	2 (3.4%)
	M_{l-p}	L_{l-t}			▨ M_{l-p}	L_{l-t}	
2	H_{n-p}	M_{n-t}	1 (1.7%)	10	▨ H_{n-p}	M_{n-t}	0
	M_{l-p}	▨ L_{l-t}			M_{l-p}	▨ L_{l-t}	
3	H_{n-p}	M_{n-t}	3 (5.1%)	11	H_{n-p}	▨ M_{n-t}	0
	▨ M_{l-p}	L_{l-t}			▨ M_{l-p}	L_{l-t}	
4	▨ H_{n-p}	M_{n-t}	0	12	H_{n-p}	▨ M_{n-t}	8 (13.6%)
	M_{l-p}	L_{l-t}			▨ M_{l-p}	▨ L_{l-t}	
5	H_{n-p}	▨ M_{n-t}	0	13	▨ H_{n-p}	M_{n-t}	0
	M_{l-p}	L_{l-t}			▨ M_{l-p}	▨ L_{l-t}	
6	H_{n-p}	M_{n-t}	0	14	▨ H_{n-p}	▨ M_{n-t}	0
	▨ M_{l-p}	▨ L_{l-t}			▨ M_{l-p}	L_{l-t}	
7	H_{n-p}	▨ M_{n-t}	12 (20.3%)	15	▨ H_{n-p}	▨ M_{n-t}	1 (1.7%)
	M_{l-p}	▨ L_{l-t}			M_{l-p}	▨ L_{l-t}	
8	▨ H_{n-p}	▨ M_{n-t}	0	16	▨ H_{n-p}	▨ M_{n-t}	24 (40.7%)
	M_{l-p}	L_{l-t}			▨ M_{l-p}	▨ L_{l-t}	

Note. H_{n-p} denotes the number-prefix method which has high situational constraint. M_{l-p} denotes the list-prefix method which has medium situational constraint. M_{n-t} denotes the number-tail method which has medium situational constraint. L_{l-t} denotes the list-tail method which has medium situational constraint.

Testing Hypothesis 3: Performance Pattern

Hypothesis 3: If participants exhibit the copies model in some but not all task situations that represent four possible combinations of call structure and parameter type, their successful performance will be clustered around tasks with higher situational constraints. Performance patterns representing such trend should occur more frequently than those contradicting it.

In the untriangulated trace data, 32 participants demonstrated hypothesis-uninformative performance patterns (6 participants for pattern 1 and 26 participants for pattern 16). These participants were not included in this analysis, because they did not demonstrate sufficient variability to permit the evaluation of performance related to situational constraint. As shown in Table 20, the aggregated observed proportions were 74.1% (20 out of 27) for the hypothesis-consistent patterns and 25.9% (7 out of 27) for the hypothesis-inconsistent patterns. An exact test of goodness-of-fit indicated a statistically significant difference between the observed distribution and the expected distribution, $p < .001$. Hypothesis-consistent patterns had 12 more occurrences than predicted by chance, and the hypothesis-inconsistent patterns had 12 less. One-sided Z-tests indicated that the aggregated proportion of hypothesis-consistent patterns was significantly higher than expected (z = 4.39, 1-tailed $p < .001$), and the aggregated proportion of hypothesis-inconsistent patterns was significantly lower (z = -4.39, 1 tailed $p < .001$).

As shown in Table 20, results using the triangulated trace data were very similar. Thirty two participants demonstrated hypothesis-uninformative performance patterns (8 participants for pattern 1 and 24 participants for pattern 16) and were excluded from this analysis. The aggregated observed proportions were 77.8% (21 out of 27) for the hypothesis-consistent patterns and 22.2% (6 out of 27) for the hypothesis-inconsistent patterns. The exact test of goodness-of-fit again indicated a statistically significant difference between the observed distribution and a random distribution, $p < .001$. Hypothesis-consistent patterns had 13 more occurrences than expected, and the hypothesis-inconsistent patterns had 13 less occurrences than expected. The Z-tests indicated that the aggregated proportion of hypothesis-consistent patterns was higher than predicted by chance (z = 5.34, 1-tailed p < .001), and the aggregated proportion of hypothesis-inconsistent patterns was lower (z = -5.34, 1 tailed p < .001).

Table 20

Z-tests for aggregated proportions of hypothesis-consistent patterns and of hypothesis-inconsistent patterns

Performance patterns	Observed frequency (proportion)	Expected frequency (proportion)	Residual	Z statistics	*p*-value (1-tailed)
		Untriangulated trace data			
Hypothesis-consistent[4]	20 (74.1%)	8 (28.6%)	12	4.39	<.001**
Hypothesis-inconsistent[5]	7 (25.9%)	19 (71.4%)	-12	-4.39	<.001**
		Triangulated trace data			
Hypothesis-consistent	21 (77.8%)	8 (28.6%)	13	5.34	<.001**
Hypothesis-inconsistent	6 (22.2%)	19 (71.4%)	-13	-5.34	<.001**

*$p < .05$, **$p < .001$

Comprehension-based Model Categorization

Participants' answers to the program comprehension question were categorized based on the simplified categorization scheme for mental models of recursion (see Chapter 3). Seventy percent (42 out of 60) of the participants correctly completed the comprehension task and were categorized as demonstrating the potential-copies model. Thirty percent (18 out of 60) of the

[4] Patterns 2, 6, 7, and 12

[5] Patterns 3, 4, 5, 8, 9, 10, 11, 13, 14, and 15

participants gave incorrect or partial answers and were categorized as demonstrating the non-copies models.

Stability of the Copies Model

Stability of the copies model was ranked by how frequently the copies model manifested itself among the four program evaluation tasks and one program comprehension task. As shown in Table 21, 6 participants (10.2% out of 59) had highly stable copies models, 5 participants' (8.5%) copies model stability was moderate high, 8 participants' (13.6%) was moderate, 11 participants' (18.6%) was moderate low, 18 participants' (30.5%) was low, and 11 participants (18.6%) failed to demonstrate the copies model in any of the five tasks.

Table 21
Stability of the copies model ranked by frequency of its manifestation

Frequency of the Copies Model Manifestation	Stability of the Copies Model	Frequency	Percentage (n=59)
In all five tasks	High	6	10.2%
In all four tasks	Moderate high	5	8.3%
In three tasks	Moderate	8	13.6%
In two tasks	Moderate low	11	18.6%
In one task	Low	18	30.5%
In none of the tasks	Absence	11	18.6%

Study 2

Identification of the Computer-as-Agent P-Prim

As an a priori theoretical construct, the computer-as-agent p-prim is identified by examining its expected expressions against three prescribed criteria: self-explanatory, functional, and context-sensitive. Specifically, mental transaction units were selected for examination if their initial contextual configurations satisfied the conditions for the computer-as-agent p-prim to activate. Then, these selected mental transaction units were coded as follows: For the self-explanatory criterion, each single mental transaction unit was coded for the participant’s emotions and linguistic features expressed within the unit. For the functional criterion, each single mental transaction unit was coded for whether it contributed to construction of the copies model of recursion. For the context-sensitivity criterion, interpretations of method invocation were coded and compared against each other within participants.

Supposition 1

Supposition 1: Interpretations generated by the computer-as-agent p-prim are characteristic of a sense of intuitive obviousness and satisfaction.

The hallmark of a p-prim’s self-explanatory characteristic is the certainty and satisfaction expressed in participants’ interpretations of computer programs. These rather subtle emotions can be detected from participants’ tone and linguistic features. When the computer-as-agent p-prim operates properly, participants’ tones are firm and clear, and they use transition words such as “so”, “then”, and “and then” to mark the completion of one operation and beginning of another one. Participants with various mental models of recursive function exhibited similar linguistic patterns.

The following two transcript segments show two participants’ evaluation processes of the same method. The first participant had a robust understanding of recursion, while the second participant had a poor understanding.

> ……(Unit 1) So first, there is the integer n, and n is the size of the list, so I know that n is 4, because there are four values in this list…(Unit 2) Then I went on to the IF statement, so it said if i is equivalent to n minus 1, so if 0 is the same as 4 minus 1, which is 3, and that's obviously not true, so we don't do the function that's under the IF statement, we go to the else statement…(Unit 3) So it says, it defines two variables, says u is list.get(i), which means the integer u is goanna be our first value in the list, so I know that u is goanna be equal to 2. (Unit 4) Then it says that v is goanna be equal the second value in the list because it's i+1, the reference value i plus 1 more, so v is goanna be 1 because that's the second value in the list……(participant No.55 evaluating the list-prefix method)

> ……(Unit 1)Well since the list size is 4, I wrote beside int n is equal to 4. So if n is equal to 4. (Unit 2) Then going to the next statement, the IF statement, 0, which is i, is not equal to 4 minus 1, because that is 3. (Unit 3) So I go to my else statement, and I get u, and u is therefore going to be first equal to 0 because it's list.get i. (Unit 4) And v is list.get up once, so v is equal to 1……(Participant No.7 evaluating the list-prefix method)

When the computer-as-agent p-prim operated unproductively, many participants still expressed certainty in their performance. For instance, participant No.7 demonstrated an active (base case result) model in her evaluation of the list-prefix method. She was positive about her decision to stop execution after returning base case result: "n is still 4, so i is actually this time equal to 3, so I just write true, and I return the result…...as long as it keeps running through it, that I think the result would equal true."

Some participants expressed some uncertainty in their performance but could not pinpoint where things went wrong. It appeared to them that there was no alternative. For instance, participant No. 14 demonstrated an active (base case result) model in her evaluation of the number-tail method. She was uncertain about her answer but was unable to point out what made her feel uncertain:

> So this is another one that I don't know how to do, but basically what it was, I started at g(10, 3) …[evaluating invocation g(10, 3), g(10, 2), and g(10, 1)]… I went back to the

> top [invocation g(10, 0)], 10 times 0 is 0, and then, 0, this IF statement does not apply anymore because n is equal to 0, and so I was like, I don't know what to do now, because I thought that would be the end of the method. (Participant No.14 evaluating the number-tail method)

Sometimes, unproductive computer-as-agent p-prim operations generate results that are inconsistent with general knowledge of programming. For example, a recursive method may not generate an output in a mistaken evaluation. This result is not consistent with the general knowledge that a method should have a purpose. This cognitive dissonance generated a need for explanations or corrections. Some participants were able to indicate dissatisfactory aspects of their performance but could not find alternative solutions. For instance, participant No.30 also demonstrated an active (base case result) model in her evaluation of the number-tail method. She knew that she needed to trace back to the original invocation like she did for the methods with the prefix call structure, but she could not find a way to do that: "…I am confused on where, like why it stopped…I know I have to get to the initial value [initial invocation], then I get confused, I plug in back, where you are supposed to do it…"

These observations establish the high cuing reliability of the computer-as-agent p-prim and further substantiate the self-explanatory characteristic.

The self-explanatory characteristic can also be observed in reverse situations where participants compare the computer's behaviors with their predictions. When the computer's behavior does not match their predictions—particularly when computer continues execution after processing the base case—participants often feel bewildered, puzzled, or confused, and they need explanations for the unanticipated occurrence.

For instance, participant No.14 demonstrated an active (base case result) model in evaluation of the number-tail method. She stopped processing after evaluating the base case. When she saw the standard answer, she could not understand why the computer would continue processing backward:

> …I did 10, 0 [g(10, 0), the base case], I got the product, which is 0, and then, yea, so I guess I exited it technically instead of keep going back up, but then, I don't get this part [pointing the processing after the base case]……I don't understand why it's printing that. (Participant No.14 reviewing the standard trace for the number-tail method)

Some participants managed to generate traces reflecting the copies model, however they did so not because they understood the mechanism but because they memorized examples from class. When asked to explain the passive flow, they admitted their lack of understanding. For instance, participant No.4 successfully demonstrated the copies model in her evaluation of the number-tail method. However, when asked how positive she was about her answer, she expressed uncertainty and indicated that she did not understand how the computer would continue executing the printing statements after processing the base case. She did so only because she remembered the examples that the teaching assistant showed right before the quiz:

> ……I wasn't very sure like…once n does equal to 0, how exactly to go about printing this statement. Like, is the rest of this [statements after the recursive call] just ignored, or, because basically it seems like we are only printing out the product [she meant calculating the product in a previous statement] and ignoring these steps [statements after the recursive call], but I was pretty I was right because we've done these steps before [in class], so. (Participant No.4 reflecting on her evaluation of the number-tail method)

Supposition 2

Supposition 2: Interpretations generated by the computer-as-agent p-prim serve participants well in many cases rather than causing problems all the time.

Functionality of the computer-as-agent p-prim is expressed in the relationship between local interpretations and global outcome. Local interpretations include the interpretations of original invocation, sub-invocation, non-recursive statement, and recursive statement. These interpretations are considered productive or unproductive depending on whether they contribute to the construction of the copies model of recursion.

Non-recursive statement. Interpretations of non-recursive statements are usually productive, because programming schemas associated with these statements are mostly structured in agentive format. For example, in an IF statement, the computer evaluates a condition to determine whether or not to execute the following section of statements. The conditional statement is the patient, and the decision is the effect. As the following excerpt shows, as soon as a decision is made, the computer-as-agent p-prim completes its operation and evaluation proceeds to the next statement: "It says once n is equal to 0, your result should print 1, so I knew this is 4, and 4 is not equal to 0, so I move to else." (Participant No.33 evaluating the number-prefix method)

Such interpretations contribute to the generation of the copies model. However, the contribution is limited, because the traces generated by these interpretations do not characterize the nested structure of the copies model.

Original invocation. Interpretations of the original invocation are mostly productive. The invocation specified in the problem statement along with the code of the method usually activates a process schema of method invocation. That is, the participants were aware that they needed to go through all the statements in the method in order to complete the evaluation task. Some participants would explicitly review each statement of the method before they evaluate the method with the given inputs, as shown in the following excerpt:

> …a new variable of integer type named product is set to equal to m times n, and if n doesn't, the exclamation means if n doesn't equal to 0, then it runs g again, or recurse the method, this time using m and n minus 1, to initialize the formal parameters, and system dot out print line, and m and so on so forth, just displaying m times n equals whatever the product may be… (Participant No.3 evaluating the number-tail method)

Such interpretation contributes to the construction of the copies model. However, it is not a critical one because evaluation can still go wrong when it comes to recursive statements and sub-invocations.

Sub-invocation. Interpretations of sub-invocations are frequently unproductive. Sub-invocations are different from original invocations in that they are specified by the participants instead of the problem statements. While the participants usually had a clear view of the original invocation, their understandings of the sub-invocations were rather vague. As the following interview excerpt shows, the participant successfully traced down to the base case, but she did not know that there were unfinished operations in each sub-invocation. Such interpretations significantly hinder the construction of the copies model:

> …so I had g of 10 comma 0, and I went back to the top, 10 times 0 is 0, and then, 0, this IF statement does not apply anymore because n is equal to 0, and so I was like, I don't know what to do now, because I thought that would be the end of the method. (Participant No.14 commenting on his trace for the number-tail method)

Recursive statement. Interpretations of a recursive statement can be productive or not depending on the nature of the statement. Usually, in prefix methods, the interpretations are productive. As the following interview excerpt shows, the statement "result = 5 * p (5, 3)" in the method p activates the computer-as-agent p-prim and a multiplication schema. Since the multiplier "p (5, 3)" is unknown, a new transaction starts to evaluate "p (5, 3)" while a pending computer-as-agent p-prim is generated for the statement "result=5 * p (5, 3)". This pending computer-as-agent p-prim contributes to the construction of the copies model because it requires the result of invocation p(5, 3) in order to complete. The following interview excerpt illustrates this mental process:

> It says result, which was initialized earlier, is equal to m, which is 5, times p of m, which is 5, comma, n minus 1, so 4 is n, so plug in 4, minus 1 is 3. But then we don't know what p of 5 comma 3 is. Then we have to go back up to the beginning… (Participant No.25 evaluating the number-prefix method)

However, the computer-as-agent p-prim can be unproductive in prefix methods when the participants incorrectly interpret the prefix operations. For instance, the statement "result = b && q (list, i+1)" in the method q is supposed to activate the logical-AND schema along with the computer-as-agent p-prim. As the operand "q(list, i+1)" is unknown, a new transaction should start to evaluate "q(list, i+1)" and a pending computer-as-agent p-prim should be generated for the statement "result = b && q (list, i+1)". However, as the following interview excerpt shows, this activation sequence breaks down because the participant misunderstands the logical-AND operator as a punctuation sign that divides a single statement into two separate statements: returning the value of variable b and invoking the method q again. Accordingly, the computer-as-agent p-prim is activated sequentially for each of the two mentally carved statements. Such an operation is apparently unproductive.

> …so return false [value of variable b is false], and the arraylist and then the index plus 1, so that was 1. Yea, I got confused here because I don't remember if it returned this or if it waited until the very end to return it, but I just had it returned anyway. But then you went back did the whole statement again… (Participant No.29 evaluating the list-prefix method)

Supposition 3

Supposition 3: Interpretations generated by the computer-as-agent p-prim are sensitive to local context, thus they are likely to change in response to change in local context.

Interpretation of method invocation is the key to construction of the copies model of recursive function, thus this analysis is focused on the inconsistent interpretations of the method invocations across three pairs of different contexts.

Inconsistent interpretations of method invocations across tasks. A method invocation, particularly a sub-invocation generated by a recursive call, has different contextual configurations in different tasks as these tasks were designed to have unique combinations of call structure and parameter type. Analysis of the interview data revealed a substantial amount of inconsistency in the participants' interpretations of sub-invocations across tasks. As shown in Table 22, in the number-prefix task these participants interpreted the sub-invocation as a product generated by a process (product-by-process schema) or a process that generates a product (process-to-product schema), whereas in the number-tail task they interpreted it as an entrance to a series of actions (entrance schema) or a process that is composed of a series of actions (process schema). A similar interpretive pattern appears between the list-prefix task and the list-tail task.

Table 22

Inconsistent interpretations of sub-invocations across tasks

Participants	Tasks			
	Number-prefix	Number-tail	List-prefix	List-tail
No.14	Product-by-process	Entrance	Product-by-process	Entrance
No.39	Product-by-process	Entrance	Entrance	Entrance
No.54	Product-by-process	Entrance	Product-by-process	Entrance
No.24	Process-to-product	Entrance	Process-to-product	Entrance
No.40	Process-to-product	Entrance	Process-to-product	Entrance
No.25	Product-by-process	Process	Product-by-process	Process
No.33	Product-by-process	Process	Product-to-process	Entrance
No.56	Product-by-process	Process	Product-by-process	Process
No.55	Process-to-product	Process	Process-to-product	Process
No.3	Process-to-product	Process	Process-to-product	Process

Inconsistent interpretations of method invocations within a single task. A method invocation also has two different contextual configurations within a single task. It can be the original invocation specified by the problem statement, or it can be the sub-invocations specified by the participants during evaluation. Analysis of the interview data showed some inconsistency in the participants' interpretations of these types of method invocations. As shown in Table 23, the participants interpreted the original invocation as a process, but for the sub-invocations they would invoke the entrance schema or the product-by-process schema.

Table 23

Inconsistent interpretations of method invocations within a single task

Participants	Contexts	
	Original invocation	Sub-invocation
No.33	Process	Product-by-process or entrance
No.14	Process	Entrance
No.54	Process	Product-by-process or entrance
No.40	Process	Entrance
No.25	Process	Product-by-process
No.24	Process	Entrance

Inconsistent interpretations of sub-invocations. A more nuanced difference in contextual configurations exists among sub-invocations within a single task. Although these invocations are surrounded by almost identical program elements, their temporal sequence varies by when they are unfolded during evaluation. Analysis of the interview data revealed some inconsistency in the participants' interpretations of these temporally different sub-invocations. As shown in Table 24, although these participants interpreted all the sub-invocations as a process upon first encounter, only the interpretations of the last one or two invocations sustained.

Table 24
Unstable interpretations of sub-invocations within a task

Participants	Sub-invocations			Tasks
	1st	2nd	3rd	
No.33	Process*	Process	Process	Number-tail
No.25	Process*	Process	Process	Number-tail and list-tail
No.3	Process*	Process	Process	Number-tail and list-tail
No.55	Process*	Process*	Process	List-tail

Note. *Unstable activation of the process schema.

Schemas of Method Invocation

A set of schemas of the Java method invocation were discovered through examination of participants' interpretations of the method invocation in various contexts. These schemas vary in their approximation to the formal concept of method invocation.

Process Schema

The process schema refers to the view of a method invocation as an integral body composed of a series of actions. Once a method is invoked, all the actions constituting the invocation must be completed. This schema sometimes explicitly manifests itself in participants' interpretations of the original invocation. For example, participant No.41

reviewed each statement of the number-tail method before tracing it: "…so we have public static void g, ah, something, something…something g m n minus 1, system dot da da da, end IF statement, and end method, so these two (brackets) go together, these two (brackets) go together…"

When interpreting the sub-invocations, the process schema usually manifests itself in implicit ways. As shown in the following excerpts, the process schema is expressed in the unproblematic transition from completion of the sub-invocation to the next statement in the calling invocation:

> …so this (recursive invocation) finally completes, then because of that, I can go back to the system, to the print system, and then complete… (Participant No.41 evaluating the number-tail method)
>
> …so then it goes back to the statement after it said this in the previous one…so that's (the recursive call) complete, and now we go to next statement in the IF statement. (Participant No.56 evaluating the number-tail method)
>
> …and then you are back to where you left off in previous method…and you reach the print line statement…you reach the end of that method, go back where you were before. (Participant No. 15 evaluating the number-tail method)
>
> …each time I am running this method, I am not getting to…the printing line…until I have got all the way to the bottom. (Participant No. 55 evaluating the number-tail method).

Process-to-Product Schema

The process-to-product schema is a variant of the process schema. It refers to the view of a method invocation as a series of actions that generate a product. It is commonly expressed in interpretations of *type methods* which indeed return a value, as the following interview transcripts show:

> …I want to multiply my m value by whatever this method would give me……so once this method returns 1, it says, alright, basically, this part of the previous method is now 1… (Participant No. 55 evaluating the number-prefix method)

> …you have to keep going back to n minus 1…keep going back until it equal 0…when you got that value…and then you will take this 5 and put it back into the recursion for 5 2, and you just keep bring this number back up here. (Participant No.39 evaluating the number-prefix method)

> …the result is goanna be the first number, which is 5, times the method, using 5 and 3 this time…now we have 5 times the method of 5 and 2…you need to plug it into this one I did…here is where you plug it into… (Participant No.40 evaluating the number-prefix method)

> …the result is goanna be 5 times p 5 3, and we need to know the result of p 5 3…the result is goanna be 5 times p 5 2, obviously we need to know the value of p 5 2… (Participant No.54 evaluating the number-prefix method)

Product-by-Process Schema

The product-by-process schema refers to the view of a method invocation as a product generated by a series of actions. It differs from the process-to-product schema in that it centers on the product rather than the process. This schema is often expressed in interpretations of type methods which return values. Sometimes, it shows in the declarative content clauses used to refer to the recursive call, as shown in the following excerpts:

> …ok I need to figure out what 5 times 3 is, so I went back through it…so I still don't know what p of 5 comma 2 was, so I gotta figure that out…I still don't know what p of 5 comma 1 was, so I put, I was like ok I need to figure that out… (Participant No.14 evaluating the number-prefix method).

> …but then we don't know what p of 5 comma 3 is, then we have to go back up to the beginning…and then you do it again, because you still don't know what p of 5 comma 1 is… (Participant No.25 evaluating the number-prefix method)

Sometimes, the product-by-process shows in the verb used with the recursive call. For example, participant No.12 said: "I was just trying to find p of 5 4…and then I got 5 times the invocation of 5 3, and then from there, I still couldn’t figure that out…and then couldn’t figure out that one yet.”

Occasionally, the product-by-process schema also manifests itself in interpretations of void methods, even though void methods do not actually return any value. As shown in the following interview excerpt, the participant believed that the void method invocations generate products that can be somehow combined together:

> …I was trying to find 10 and 3……and then you take…g of m and n-1…so that would be those 10 time 2 is 20…then you would be 10 times 1……so I have 20, and from 20 I get 10 and 0, and then I don’t understand how they would work exactly… (Participant No.12 evaluating the number-tail method)

Entrance Schema

The entrance schema refers to the view of a method invocation as an entrance to a series of actions. It fundamentally differs from the process schema. In this entrance schema the actions may be completed or not depending on where the exit is located, whereas in the process schema all the actions must be completed because the actions constitute an integral body. The expressions of this entrance schema are subtle. The participants use informal phrases such as “do”, “go back to”, and “take” to describe invoking of a method, and their descriptions convey a sense of locating the next statement to be evaluated rather than invoking a method as a whole process:

> …then you do g of 10 comma 2…so then I would do g of 10, 1, so go back to the top…so I had g of 10 comma 0, and I went back to the top… (Participant No. 14 evaluating the number-tail method)

> …you have to keep going back to when n equals 2, and when n equals 1… (Participant No.39 evaluating the number-tail method)

> …so we are going to take 10 and 2, up here again…so we go back to the top, now we are working with 10 1…do this again, but with 1 less than 1… (Participant No.40 evaluating the number-tail method)

Variable-Updater Schema

The variable-updater schema refers to the view of a method invocation as updating values of the variables held in the parentheses. It is occasionally expressed in the interpretations of the void methods in which a recursive call alone constitutes a statement. As the following excerpts show, the participants took the “updated” the variable values to the printing statement following the recursive call:

> ……so n wasn’t equal to 0, so I just, it becomes g 10 and then 3-1, so g 10 2, so after you’ve done that statement, it goes ahead and print this out…so that would be m which is 10, times, n which is 2, and then it wanted the product…(Participant No.29 evaluating the number-tail method)
>
> …invoke the statement of g m n minus 1…m will always be 10, but n minus 1 would be 2, and you should print out the statement of 10 times 2 equals to product of 20…(Participant No.7 evaluating the number-tail method)
>
> *Interviewer*: you got product m times n, which is 30, right? And I would assume that you are goanna to print out 10 multiply 3 equals to 30, so tell me why you wrote down 2 instead? Participant: did it change right here? Subtract by 1? (Participant No.34 evaluating the number-tail method)

Mathematical-Parentheses Schema

The mathematical-parentheses schema refers to the view of the method invocation as a mathematical operation in which the parentheses represent order of operation. This schema is occasionally expressed in interpretations of the method invocation with a prefix operation. As the following interview excerpt shows, participant No.34 mistakenly evaluated the recursive statement as if performing mathematical calculation: “…since you get n minus 1, you get 3, so

I added them, then multiply them to reach 40."

Relationship with the Formal Concept of Method Invocation

Among these schemas (summarized in Table 25), the process schema is equivalent to the formal concept of method invocation, while the mathematical-parentheses schema is completely out of touch with this concept. Other schemas represent different aspects of the formal concept of method invocation. The process-to-product schema and product-by-process schema only apply to the type methods which return values. The entrance schema only takes the "invoking" facet of method invocation. The variable-updater takes the "changing parameter value" facet of method invocation.

Table 25
Schemas of method invocation

Schemas	Schematization	Operation
Process	an integral body composed of a series of actions	Work through a series of actions from beginning to the end
Process-to-product	a series of actions that generate a product	Work through a series of actions from the beginning to the end and obtain a product
product-by-process	a product generated by a series of actions	Obtain a product by working through a series of actions
Entrance	an entrance to a series of actions	Enter a series of actions, may exit in the middle if an exit is present
Variable-updater	value changing operation	Update values of the variables held in the parentheses
Parentheses	Part of a mathematical operation in which the parentheses represent order of operation	Imposes order of operation to the assumed mathematical operation

Other Model Relevant Programming Schemas

Many participants also had misunderstandings of some of the non-recursive function elements. Most of these misunderstandings do not affect the construction of the copies model. For instance, several participants misunderstood the method “u = list.get (i)” as assigning variable u with the value of variable i. This mistake may lead to a wrong answer but does not affect the structure of a trace. There is only one instance where a participant’s misunderstanding of a non-recursive function element potentially affected whether she would construct the copies model. This problematic element is the logical-AND operator (i.e., double ampersand “&&”) in the list-prefix method. Because one of its operands is the recursive call, when participants correctly understand its meaning, they infer that the recursive call should return a Boolean value (i.e., the product-by-process schema). However, when they misunderstand it, they lose the opportunity to infer what the recursive call represents. As a result, they may invoke unproductive schemas of method invocation.

The alternative schemas of logical-AND operator are listed in Table 26. The connection schema is a view of the logical-AND operator as a function to connect two elements for display. The punctuation schema is a view of the logical-AND operator as a punctuation sign between two sub-statements. Neither of these two alternative understandings can remind the participants that the recursive call will return a Boolean value.

Table 26

Alternative schemas of the logical-AND operator

Schemas	&& represents…	Actions generated by the schema
Logical And	The logical-AND operator	Check whether both operands are true
Connection	Connection between two elements	Connect two elements in display
Punctuation	Punctuation between two statements	Complete the first half of a statement and then move onto the other half.

Coordination between Computer-as-Agent and Process-related Schemas

Three major modes of coordination between the computer-as-agent p-prim and the process-related schemas—unconditional coordination, conditional coordination, and incoordination—emerged from examining knowledge activation patterns throughout and across the four tasks with varying levels of situational constraint. These modes are differentiated by two mental transaction features: 1) whether instantiated computer-as-agent p-prim activates the process-related schemas; 2) if it does, whether the activation is unconditional or relies on contextual configuration shaped by the task features. The unconditional coordination mode and conditional coordination mode also have subcategories differentiating the stability of coordination between the computer-as-agent p-prim and the process-related schema. If coordination persists through the evaluation process, the mode is labeled as stable. Otherwise, the mode is labeled as unstable. The following sections present three sets of detailed description of each mode and analysis of critical mental transaction units for representative cases.

Incoordination Mode

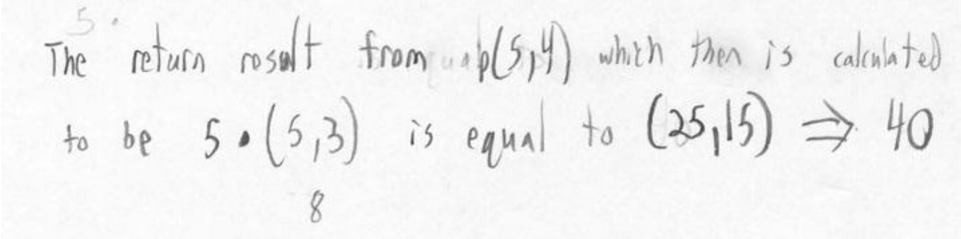
Incoordination mode describes the following situation: the computer-as-agent p-prim does not activate the process-related schemas in all tasks or only does so in the high-constraint task. It either coordinates with the entrance schema or other idiosyncratic schemas of method invocation. Table 27 shows the 9 participants whose coordination modes are categorized as incoordination.

Table 27
Incoordination mode—lack of coordination between the computer-as-agent p-prim and the process-related schemas

Participants	Tasks			
	Number-prefix	List-prefix	Number-tail	List-tail
No.34	Other	Other	Other	Other
No.16	Entrance	Entrance	Entrance	Entrance
No.60	Process*	Entrance	Other	Other
No.58	Process*	Entrance	Other	Other
No.29	Process*	Entrance	Other	Other
No.57	Process*	Other	Entrance	Other
No.42	Process*	Entrance	Entrance	Entrance
No.24	Process*	Entrance	Other	Entrance
No.7	Process*	Entrance	Other	Entrance

Note. * denotes unstable coordination

Participant No. 16 exemplifies a special case in which the computer-as-agent p-prim consistently activates the entrance schema for all tasks. Participant No.34 exemplifies another special case in which the computer-as-agent p-prim only activates other schemas even in the high-constraint task. For instance, when participant No.34 evaluated the recursive call in the number-prefix method, he simply ignored the method name and turned the recursive call into a mathematical operation (see Figure 9). It was clear that this mistake was not due to an oversight of the method name, because he had consistently misinterpreted the recursive call throughout the quiz.



5 •
The return result from uap(5,4) which then is calculated
to be 5 • (5,3) is equal to (25,15) ⇒ 40
8

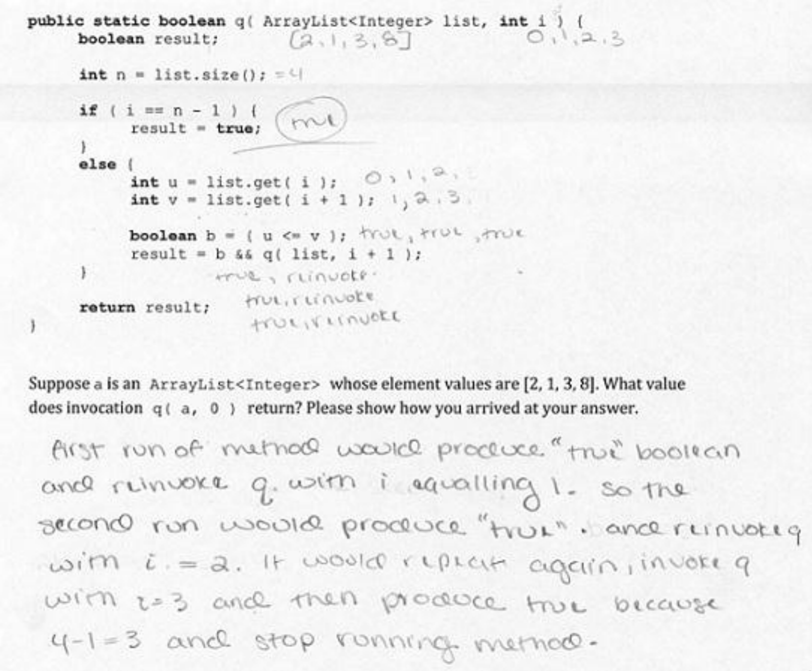
Figure 9. Participant No.34’s evaluation of the number-prefix method.

For most participants in this category, the computer-as-agent p-prim only unstably activates the process-related schemas in the high-constraint task, and it tends to activate the entrance or other schemas in medium- or low-constraint tasks. The following section presents analysis of critical mental transaction units for a case representative of this kind.

Overview of case No.7. Participant No.7’s traces are categorized as a bottom up model, a step model, an active (output from all) model, and an active (output from all) model for the number-prefix method, number-tail method, list-prefix method, and list-tail method, respectively. Her interview transcript, however, suggests that she meant to demonstrate the active (base case result) model for the list-prefix method. Either way, she failed to demonstrate the copies model in any of the tasks, suggesting failed coordination between the computer-as-agent p-prim and the process-related schemas. To illustrate, the following two sections present two sets of mental process description and analysis of the critical mental transaction units, one for the list-prefix method, and the other for the list-tail method.

Incoordination in medium-constraint task. Participant No.7’s evaluation of the list-prefix list-prefix method was incorrect. As shown in Figure 10, her answer was “true”. Her trace was categorized as an active (base case result) model (See

Appendix 5 for the complete transcript). She successfully evaluated the statements above the recursive call except a minor misunderstanding of the method list.get (i) and list.get(i+1). Instead of obtaining the list's elements whose indices are i and i+1, she mistakenly obtained the given values of i and i+1 for variable u and variable v. When she moved down to the statement involving the recursive call "result=b&&q(list, i+1)", her initial report was unclear as to her intention with the double ampersand sign: "And I report the result, true, and also reinvoke the method of q list i plus 1. And then you go back down." Her report about this code line in the second invocation clarified her intention: "And then I reinvoke the statement because the result is true, reinvoking the list i plus 1." She completely ignored the logical-AND operator and assumed that this statement would reinvoke the method q when b was evaluated to true. She proceeded to the last invocation q(a, 3), correctly evaluated the IF statement, determined the result as true, and then decided to exit the function.



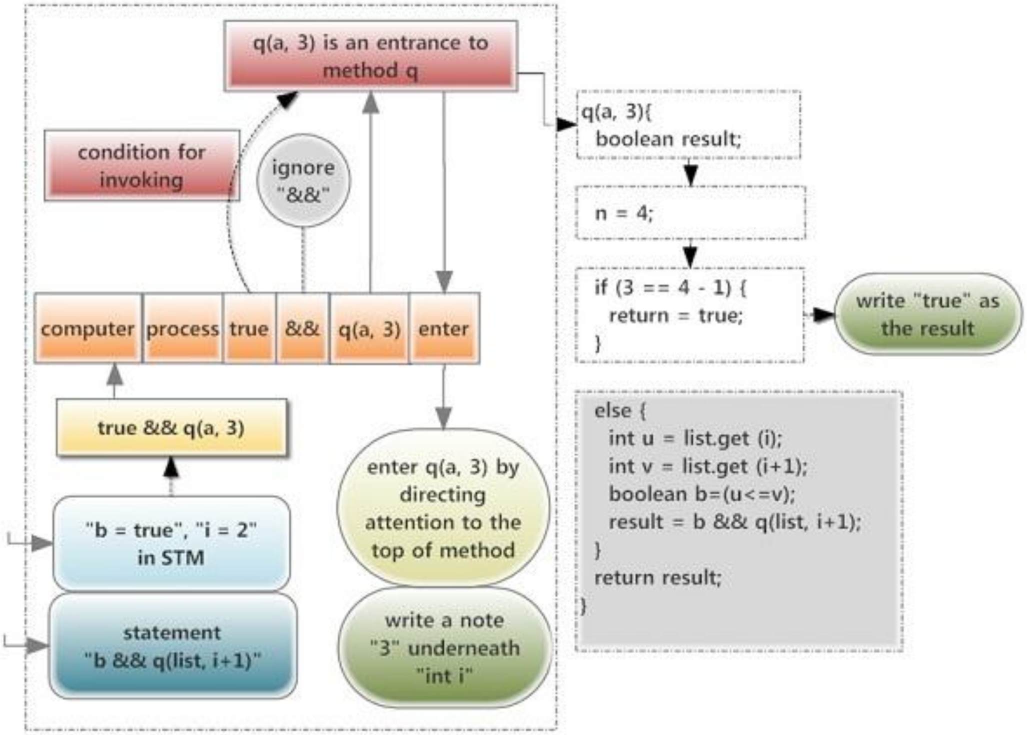
```
public static boolean q( ArrayList<Integer> list, int i ) {
     boolean result;        [2,1,3,8]              0,1,2,3

     int n = list.size(); =4

     if ( i == n - 1 ) {
          result = true;   true
     }
     else {
          int u = list.get( i );   0,1,2,
          int v = list.get( i + 1 );   1,2,3

          boolean b = ( u <= v ); true, true ,true
          result = b && q( list, i + 1 );
     }          true, reinvoke.

     return result;   true,reinvoke
                       true,reinvoke
}
```

Suppose a is an `ArrayList<Integer>` whose element values are [2, 1, 3, 8]. What value does invocation `q( a, 0 )` return? Please show how you arrived at your answer.

first run of method would produce "true" boolean
and reinvoke q. with i equalling 1. So the
second run would produce "true". and reinvoke q
with i = 2. It would repeat again, invoke q
with i=3 and then produce true because
4-1=3 and stop running method.

Figure 10. Participant No.7's trace for the list-prefix method.

Despite her misunderstanding of the method list.get(i) and ignorance of the logical-AND operator, her thought process represents a typical active model of recursion. The three mental transaction units processing the statement "result = b && q(list, i+1)" in the first three invocations were critical to generating the active model. *Figure 11* shows a schematic diagram of one of these critical units. At the beginning of the q(a, 3) mental transaction unit, she focused on the statement "b&&q(list, i+1)". Meanwhile, she held in her short-term memory the information that the value of variable b was true and the value of variable i was 2. These three pieces of information were quickly transformed to a sensory input "true && q(a, 3)", which in turn activated and instantiated the computer-as-agent p-prim. The logical-AND operator was completed ignored. One of the operands "true" activated the "conditional" schema. The other operand "q(a, 3)" activated the entrance schema. The former allowed the latter to operate and fed p-prim's result slot with "enter". As a result, she redirected her attention to the top of method q to enter invocation q(a, 3). This mental transaction unit did not leave any pending p-prim operation. Accordingly, she decided to exit the function after determining that the IF statement in invocation q(a, 3) was true and the result was true. Throughout the evaluation process, her computer-as-agent p-prim was unconditionally coordinated with the entrance schema. There was no sign of unconditional or conditional coordination with the process related schemas.


q(a, 3) is an entrance to method q
condition for invoking
ignore "&&"
computer
process
true
&&
q(a, 3)
enter
true && q(a, 3)
"b = true", "i = 2" in STM
statement "b && q(list, i+1)"
enter q(a, 3) by directing attention to the top of method
write a note "3" underneath "int i"
write "true" as the result

```
q(a, 3){
  boolean result;
```

```
  n = 4;
```

```
  if (3 == 4 - 1) {
    return = true;
  }
```

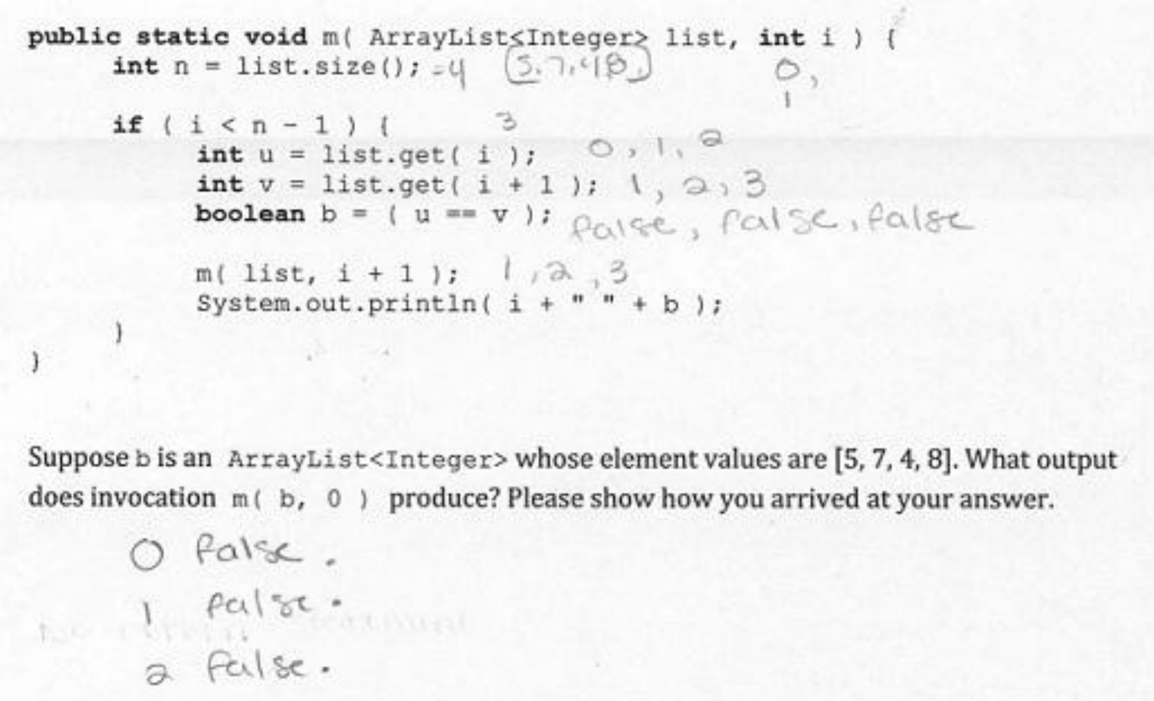
```
  else {
    int u = list.get (i);
    int v = list.get (i+1);
    boolean b=(u<=v);
    result = b && q(list, i+1);
  }
  return result;
}
```

Figure 11. Schematic diagram of participant No.7's mental transaction unit of "true&&q(a, 3)".

Incoordination in low-constraint task. No.7's evaluation of the list-tail method was incorrect. incorrect. As shown in Figure 12, her answer was "0 false, 1 false, 2 false". Her trace was categorized as active (output from all) model (See

Appendix 5 for the complete transcript). She successfully evaluated the statements above the recursive call except a minor misunderstanding of the method list.get (i) and list.get(i+1). Instead of obtaining the list’s elements whose indices are i and i+1, she mistakenly obtained the given values of i and i+1 for variable u and variable v. When she moved down to the statement m(list, i+1), her initial report was unclear as to her intention: “And then m list i plus 1, it would give you 1.” She passed the recursive call, moved down to the printing statement, and predicted what the computer would print for this invocation. At this moment, she did not recognize that she confused the sequence of the recursive call and the printing statement. Then she glanced at her trace and said: “And I don’t know why I made it run through it again. I wrote out the other two values down.”

She appeared to realize that invoking the method after the printing statement did not agree with the codes. Then she continued to explain that the computer would print the value of i and the value of b for each invocation. The value of i incremented by 1 every time, and the value of b was false all the time. As she moved to invocation m(b, 3), she correctly evaluated the IF statement as false and decided that the execution would end at this point.



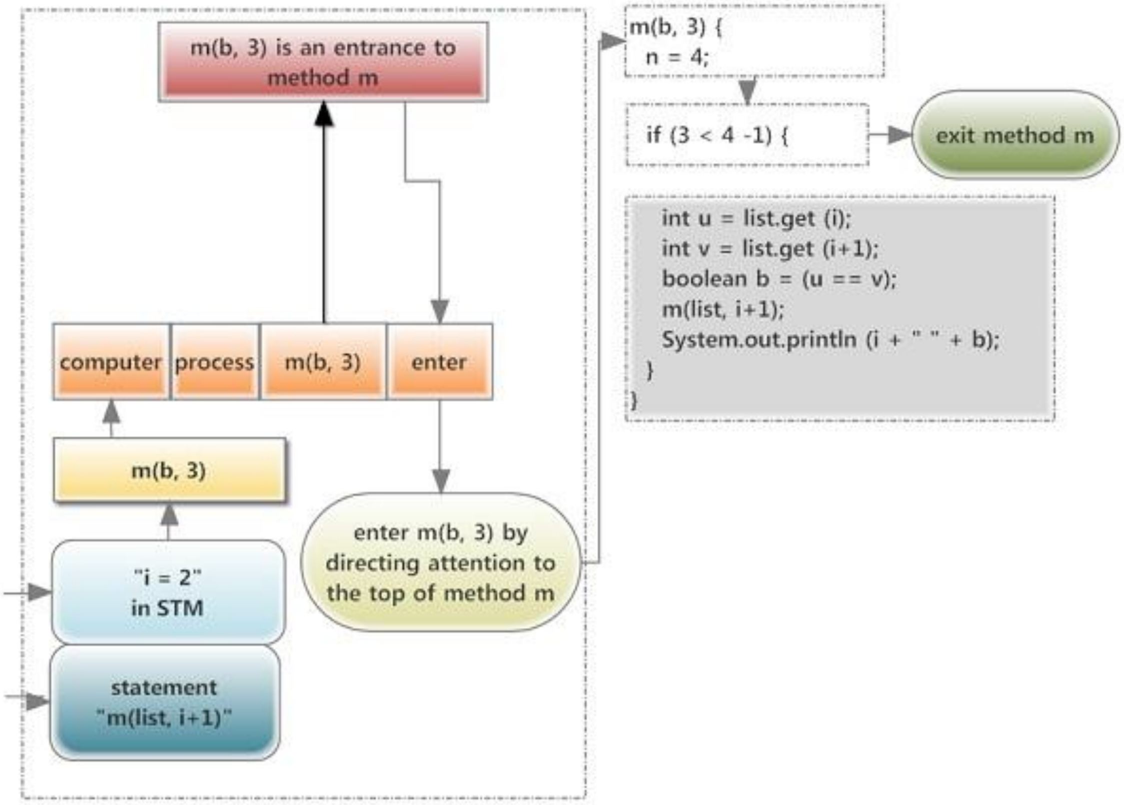
```
public static void m( ArrayList<Integer> list, int i ) {
      int n = list.size(); =4  (5,7,4,8)          0,
                                                  1
      if ( i < n - 1 ) {         3
            int u = list.get( i );    0, 1, 2
            int v = list.get( i + 1 );  1, 2, 3
            boolean b = ( u == v );  false, false, false

            m( list, i + 1 );   1, 2, 3
            System.out.println( i + " " + b );
      }
}
```

Suppose b is an `ArrayList<Integer>` whose element values are [5, 7, 4, 8]. What output does invocation `m( b, 0 )` produce? Please show how you arrived at your answer.

0 false.
1 false.
2 false.

Figure 12. Participant No.7's evaluation trace for the List-tail method.

Despite her misunderstanding of the method list.get(i) and confusion of the code sequence, her thought process represents a typical active model of recursion. The three mental transaction units processing the statement m(list, i+1) in the first three invocations were critical to generating the active model. Figure 13 shows a schematic diagram of one of these critical units. At the beginning of the m(b, 3) mental transaction unit, she focused on the statement "m(list, i+1)". Meanwhile, she held in her short-term memory the information that the value of variable i was 2. These two pieces of information were quickly transformed to a sensory input m(b, 3), which in turn activated and instantiated the computer-as-agent p-prim. The patient component "m(b, 3)" activated the entrance schema, which in turn fed p-prim's result slot with "enter". As a result, she redirected her attention to the top of method m to enter invocation m(b, 3). This mental transaction unit did not leave any pending p-prim operation. Accordingly, she decided to exit the function after determining that the IF statement in invocation m(b, 3) was false. Throughout the evaluation process, her computer-as-agent p-prim was unconditionally coordinated with the entrance schema. There was no sign of unconditional or conditional coordination with the process-related schemas.


m(b, 3) is an entrance to method m
computer
process
m(b, 3)
enter
m(b, 3)
"i = 2" in STM
statement "m(list, i+1)"
enter m(b, 3) by directing attention to the top of method m
exit method m

```
m(b, 3) {
  n = 4;
  if (3 < 4 -1) {
    int u = list.get (i);
    int v = list.get (i+1);
    boolean b = (u == v);
    m(list, i+1);
    System.out.println (i + " " + b);
  }
}
```

Figure 13. Schematic diagram of participant No.7's mental transaction unit m(b, 3).

Conditional Coordination Mode

Conditional coordination mode describes the following situation: The computer-as-agent p-prim does not unconditionally coordinate with the process-related schemas of method invocation. Specifically, pending p-prim operation is absent in medium- or low-constraint tasks. Coordination occurs only in certain situations which can shape contextual configuration to promote the activation of the process-related schemas. Thus, the categorization criterion: in addition to coordination in the high-constraint task, coordination also occurs in one or two but not all of the medium- or low-constraint tasks. The coordination is considered unstable if it does not persist through in two or more tasks.

Table 28 shows 9 participants whose coordination modes are categorized as conditional coordination. Five of them also showed unstable coordination. The following section presents analysis of critical mental transaction units for a case representative of this category.

Table 28
Conditional coordination mode

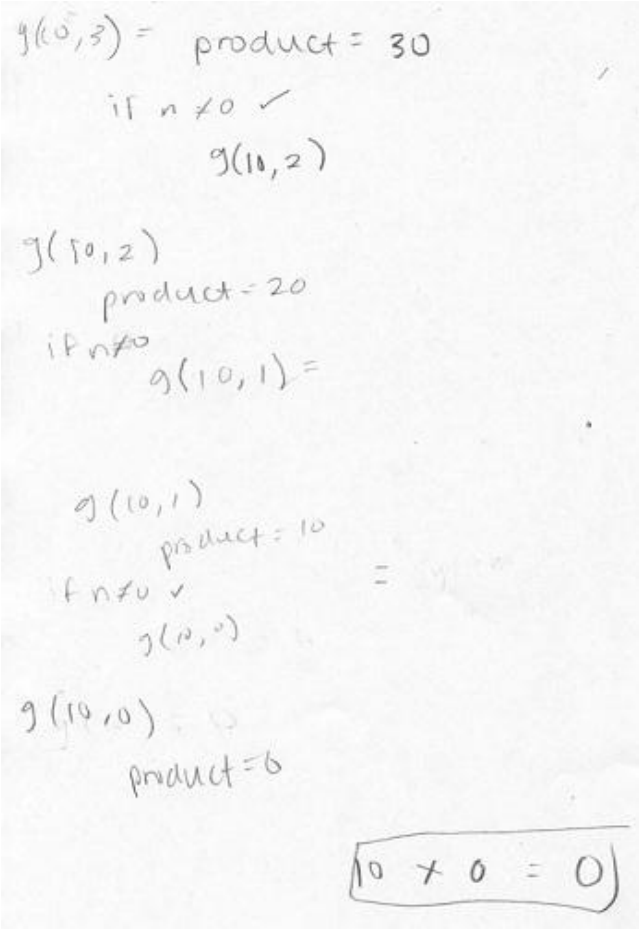
Participants	Tasks			
	Number-prefix	List-prefix	Number-tail	List-tail
No.55	Process	Process	Process	Entrance
No.14	Process	Process	Entrance*	Process*
No.12	Process	Process	Process*	Entrance
No.54	Process	Process	Process*	Entrance
No.39*	Process	Process*	Process*	Entrance
No.30*	Process*	Process*	Entrance	Process*
No.6*	Process*	Process*	Entrance	Process*
No.13*	Process*	Process*	Entrance	Entrance
No.10*	Process*	Entrance	Process*	Process*

Note. * denotes unstable coordination.

Overview of case No.14. Participant No.14's traces were initially categorized as copies model, active (base case result) model, shortcut model, and active (revert to second last) model for the number-prefix method, number-tail method, list-prefix method, and list-tail method, respectively. Her interview transcript, however, suggested that she would have demonstrated the copies model for the list-prefix method if shortcut evaluation was not allowed.

Participant No14's performance pattern is in line with the hypothesis 1 made based on the computer-as-agent p-prim conjecture. She demonstrated the copies model in the two prefix methods but not in the two tail methods. The reason for the inconsistent performance seems to lie in the different levels of constraint that the two types of call structure impose on the computer-as-agent p-prim. To illustrate this point, the following two sections present two sets of mental process description and analysis of the critical mental transaction units, one for the number-prefix method, and the other for the number-tail method.

Conditional condition in medium-constraint task. No.14's evaluation of the number-tail method was incorrect. As shown in *Figure 14*, her answer appeared on the quiz sheet was "10*0=0". She explained in the interview that she intended to output nothing. Either way, her thought process would be categorized as active (base case result) model (See Appendix 6 for the complete transcript). She successfully evaluated all the statements above the recursive call. When she moved down to the recursive call, she also correctly invoked the method with new parameter values. However, when she got to the IF statement in the last invocation, she decided to exit the function without returning to the previous invocations: "And then 0, this IF statement does not apply anymore because n is equal to 0. So I was like, I don't know what to do now, because I thought that would be the end of the method."



g(10,3) = product = 30

if n ≠ 0 ✓

g(10,2)

g(10,2)

product = 20

if n≠0

g(10,1) =

g(10,1)

product = 10

if n≠0 ✓

=

g(10,0)

g(10,0)

product = 0

$\boxed{10 \times 0 = 0}$

Figure 14. Participant No.14's trace for the number-tail method.

The three mental transaction units processing the recursive call in the first three invocations were critical to generating the active model. *Figure 15* shows a schematic diagram of one of these critical units. At the beginning of the g(10, 0) mental transaction unit, she focused on the note "g(10, 0)" that she wrote after evaluating the IF statement in invocation g(10, 1). This note directly became the sensory input, which activated and instantiated the computer-as-agent p-prim. The patient component "g(10, 0)" activated the entrance schema, which in turn fed p-prim's result slot with "enter". As a result, she wrote another "g(10, 0)" on a separate line and redirected her attention to the top of method g to enter invocation g(10, 0). This mental transaction unit did not leave any pending p-prim operation. Therefore, she decided to exit the function after determining that the IF statement in invocation g(10, 0) was false. Throughout the evaluation process, her computer-as-agent p-prim was unconditionally coordinated with the entrance schema. There was no sign of unconditional or conditional coordination with the process-related schemas.

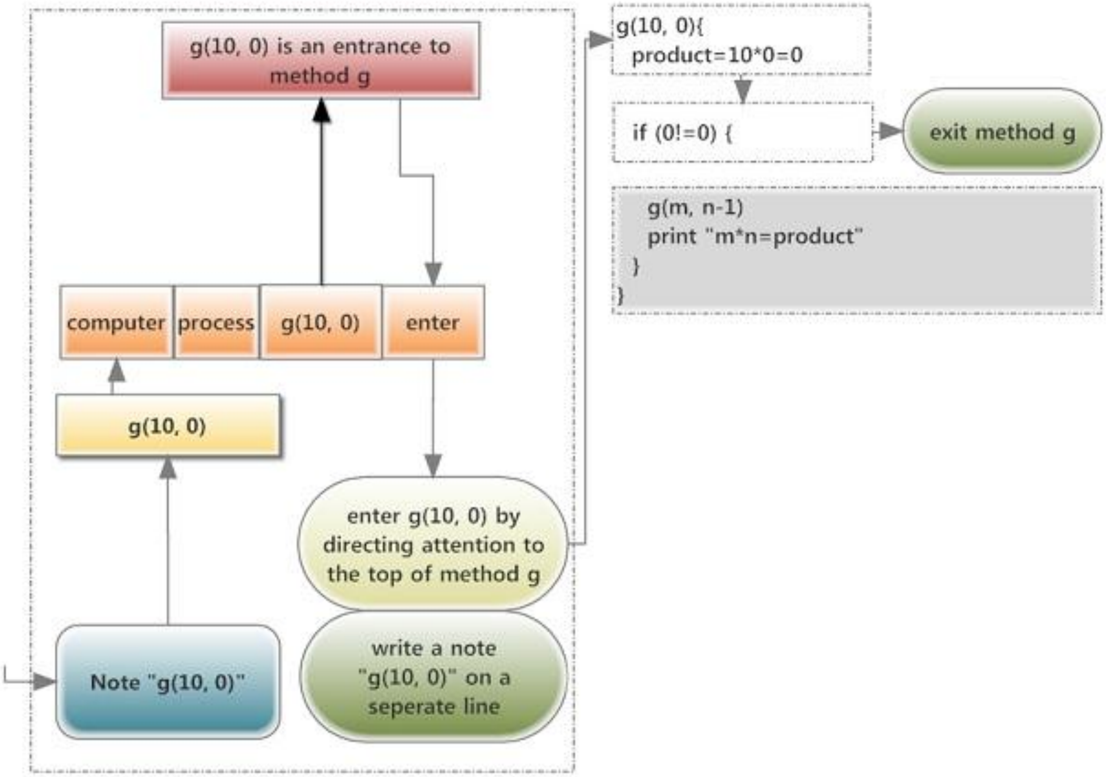

g(10, 0) is an entrance to method g
g(10, 0){
product=10*0=0
if (0!=0) {
exit method g
g(m, n-1)
print "m*n=product"
}
}
computer
process
g(10, 0)
enter
g(10, 0)
enter g(10, 0) by directing attention to the top of method g
Note "g(10, 0)"
write a note "g(10, 0)" on a seperate line

Figure 15. Schematic diagram of participant No.14's mental transaction unit g(10, 0).

Conditional coordination in high-constraint task. No.14's evaluation of the number-prefix method was correct. As shown in *Figure 16*, her trace was categorized as the copies model. Her interview transcript also confirmed the copies model manifestation (See Appendix 6 for the complete transcript). Every time she encountered the statement involving the recursive call, she particularly noted the need to obtain the value of the recursive call before processing the statement:

> Then I got else [statement]. The result equals m*p(m, n-1), so that would be, so for p(5, 4) would be 5 * p(5, 3). So then I was like, OK, I need to figure out what p(5, 3) is. So I went back through it……So I still don't know what p(5, 2) is, so I gotta figure that out……I still don't know what p(5,1) was, so I put, I was like, OK, I need to figure that out……And then p(5,0). (Participant No.4 evaluating the number-prefix method)

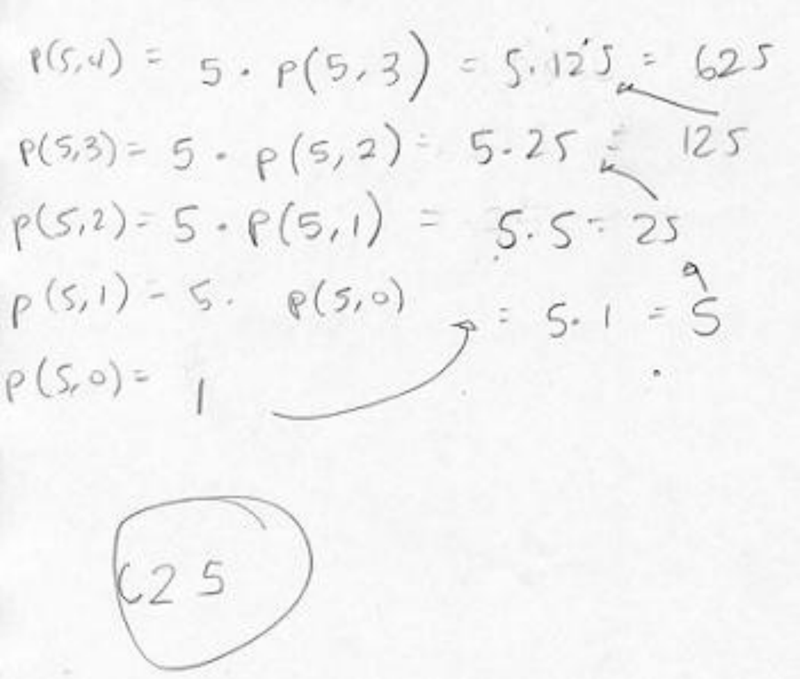
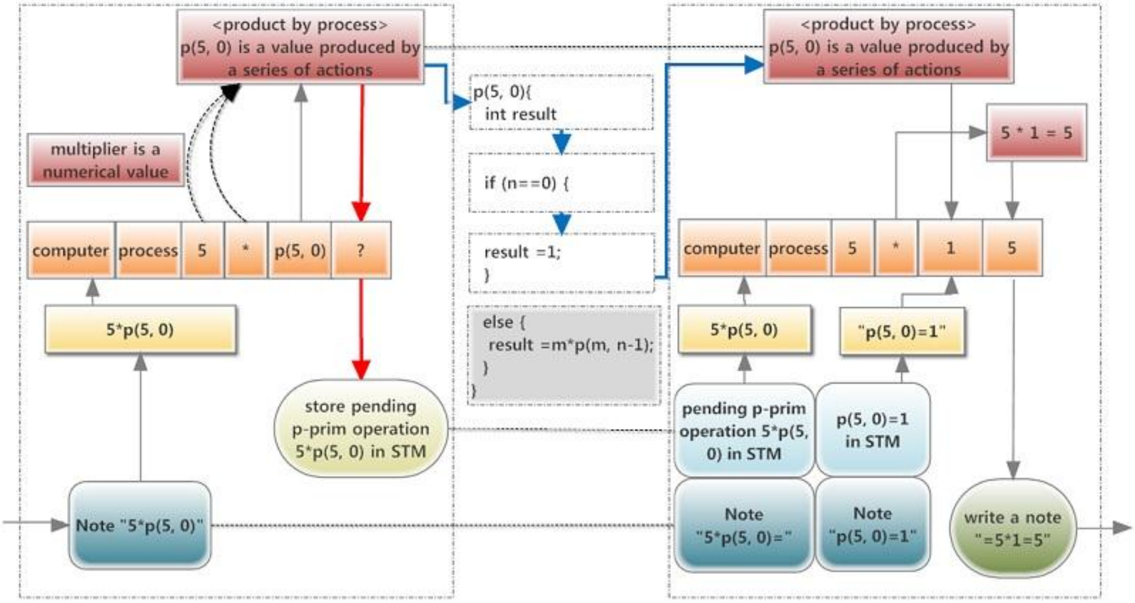

p(5,4) = 5 · p(5,3) = 5·125 = 625
p(5,3) = 5 · p(5,2) = 5·25 = 125
p(5,2) = 5 · p(5,1) = 5·5 = 25
p(5,1) = 5 · p(5,0) = 5·1 = 5
p(5,0) = 1
625

Figure 16. Participant No.14's evaluation trace for the number-prefix method.

The three mental transaction units processing the statement m*p(m, n-1) in the first three invocations were critical to generating the copies model. *Figure 17* shows a schematic diagram of the last unit 5*p(5, 0). At the beginning of this unit, she focused on the note "5*p(5, 0)" that she wrote after evaluating the IF statement in invocation p(5, 1). This note directly became the sensory input, which activated and instantiated the computer-as-agent p-prim. The patient component features a multiplication operator and two operands "5" and "p(5, 0)". The multiplication operator activated the schema that multipliers must be numerical values. This active schema shaped the contextual configuration in the way that the unknown operand p(5, 0) was likely to activate a schema aligned with the prescribed category (i.e., numerical value). In this case, p(5, 0) activated the product-by-process schema, which means the No.14 viewed the recursive call as a value produced by a series of actions. This product-by-process schema could not fill p-prim's effect slot, so it required actions to diverge. On one hand, she stored the pending p-prim operation in her short-term memory. On the other hand, she embarked on evaluation of invocation p(5, 0) to obtain its value. Once she got the value of p(5, 0), application of the product-by-process schema was complete, which allowed the pending p-prim operation to reactivate and reinstantiate with updated information. The p-prim operation completed with a simple multiplication. Then she wrote a note "=5*1=5" beside the previous note "5*p(5, 0)".


<product by process>
p(5, 0) is a value produced by a series of actions
multiplier is a numerical value
computer
process
5
*
p(5, 0)
?
5*p(5, 0)
store pending p-prim operation 5*p(5, 0) in STM
Note "5*p(5, 0)"
<product by process>
p(5, 0) is a value produced by a series of actions
5 * 1 = 5
computer
process
5
*
1
5
5*p(5, 0)
"p(5, 0)=1"
pending p-prim operation 5*p(5, 0) in STM
p(5, 0)=1 in STM
Note "5*p(5, 0)="
Note "p(5, 0)=1"
write a note "=5*1=5"

```
p(5, 0){
 int result
if (n==0) {
result =1;
}
else {
 result =m*p(m, n-1);
}
}
```

Figure 17. Schematic diagram of participant No.14's mental transaction unit for "5*p(5, 0)".

Unconditional Coordination Mode

Unconditional coordination mode describes the following situation: The computer-as-agent p-prim unconditionally coordinates with the process-related schemas.

Specifically, pending p-prim operation is present in tasks with various levels of situational constraint. Thus, the categorization criterion is that coordination occurs in all the tasks. The coordination is considered unstable if it does not persist through in one or more tasks.

Table 29 shows 10 participants whose coordination modes are categorized as unconditional coordination. Four of them also showed unstable coordination. The following section presents analysis of critical mental transaction units for a case representative of this category.

Table 29
Unconditional coordination mode

Participants	Tasks			
	Number-prefix	List-prefix	Number-tail	List-tail
No.4	Process	Process	Process	Process
No.15	Process	Process	Process	Process
No.41	Process	Process	Process	Process
No.52	Process	Process	Process	Process
No.53	Process	Process	Process	Process
No.56	Process	Process	Process	Process
No.3*	Process	Process	Process*	Process*
No.25*	Process	Process	Process*	Process*
No.40*	Process	Process	Process*	Process*
No.33*	Process*	Process	Process*	Process*

Note. * denotes unstable coordination

Overview of case No.56. Participant No.56's traces were all categorized as copies model. Her interview transcript also confirmed the model categorization. The reason for her consistent performance seems to lie in unconditional coordination between the computer-as-agent p-prim and the process schema. To illustrate this point, the following two sections present two sets of mental process description and analysis of the critical mental transaction units, one for the number-prefix method, and the other for the number-tail method.

Unconditional coordination in medium-constraint task. No.56's evaluation of the number-tail method was correct. As shown in *Figure 18*, her trace was categorized as the copies model. Her interview transcript also confirmed the copies model manifestation (See Appendix 7 for the complete transcript). She invoked the method every time she encountered the recursive call. After correctly evaluating the base case, she clearly returned to the printing statement in the previous invocation: "So then it goes back to, uh, the [printing] statement after it said this [g(10, 0)] in the previous one, so this one [pointing to g (10, 1)]." She further explained this decision: "When g(10, 1) said to go to g(10, 0), we did that, so that's complete, and now we go to next statement in the IF statement." It is clear that she viewed g(10, 1) as a process comprised of a series of actions, the major actions being the recursive call and the printing statement. She repeated this step until she returned to the original invocation and printed the last line "10*=30".

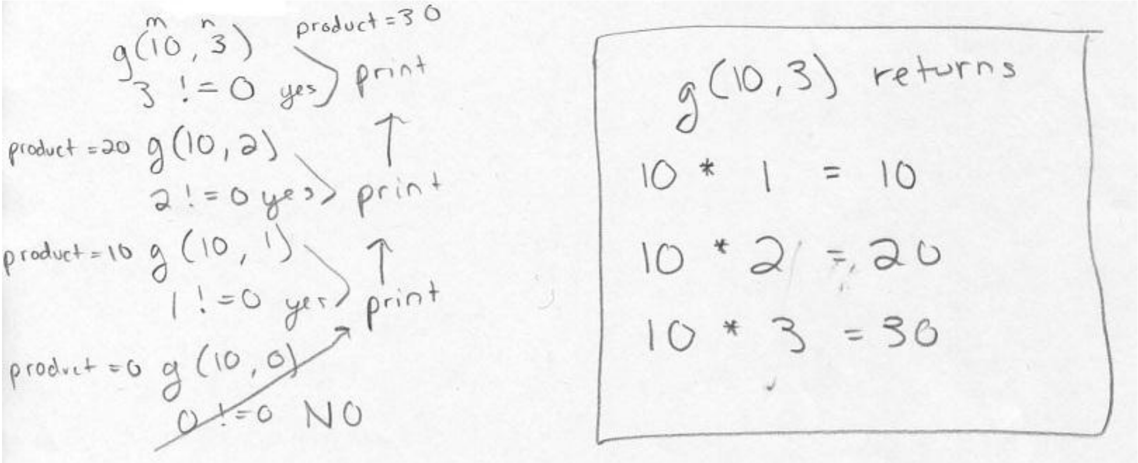


Figure 18. Participant No.56's evaluation trace for the number-tail method.

The three mental transaction units processing the recursive call in the first three invocations were critical to generating the copies model. *Figure 19* shows a schematic diagram of the last unit g(10, 0). At the beginning of this unit, she focused on the note “g(10, 0)” that she wrote after evaluating the IF statement in invocation g(10, 1). This note directly became the sensory input, which activated and instantiated the computer-as-agent p-prim. The patient component “g(10, 0)” unconditionally activated the process schema, which means the participant viewed the recursive call as a series of actions. This process schema could not fill p-prim’s effect slot, thus requiring actions to diverge. On one hand, she stored the pending p-prim operation in her short-term memory. On the other hand, she embarked on evaluation of invocation g(10, 0). Once she completed evaluation of g(10, 0), application of the process schema was complete, which allowed the pending p-prim operation to reactivate and reinstantiate with updated information.

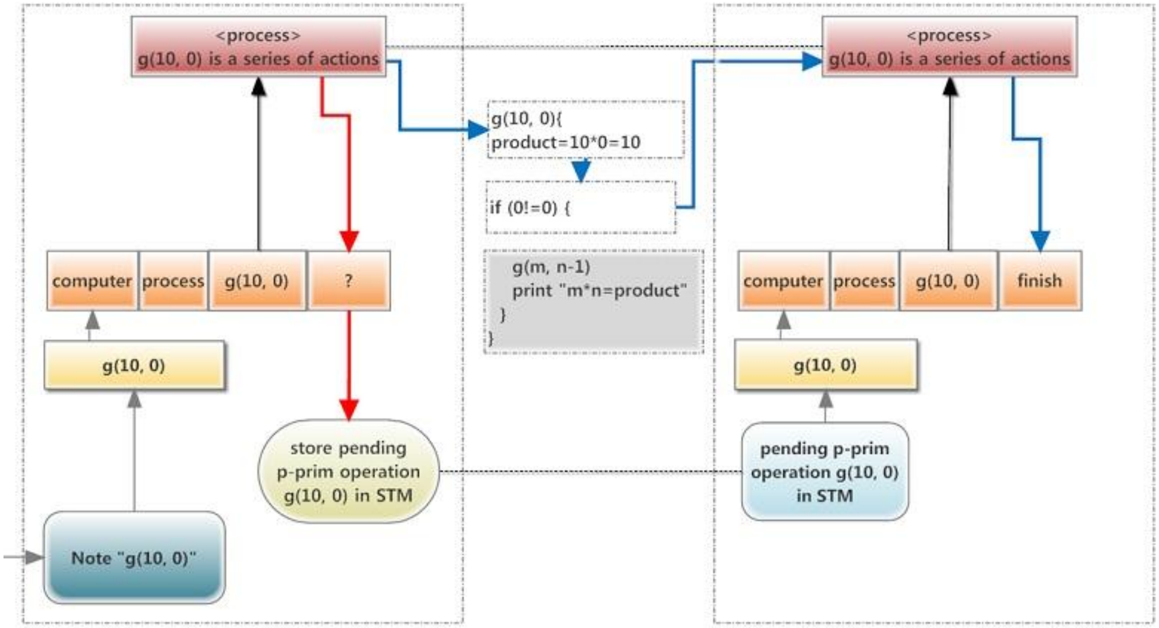


Figure 19. Schematic diagram of participant No.56’s mental transaction unit of “g(10, 0)”.

Unconditional coordination in high-constraint task. Participant No.56’s evaluation of the number-prefix method was also correct. As shown in *Figure 20*, her trace was categorized as the copies model. Her interview transcript also confirmed the copies model manifestation (See Appendix 7 for the complete transcript). She invoked the method every time she encountered the recursive call: “So then I had an invocation that said p (5, 3). And since I don’t know what p (5, 3) is, I had to go back and do the same steps I just did.” After correctly evaluating the base case, she returned to previous invocations and completed the calculation: “So now I have result equal to 1, and that means I can go back and plug in p(5, 0) into equation above it where I used it in p(5, 1).”

Although participant No.56 also viewed the recursive call as representing a value or product as participant No.14 did, there is a subtle difference between the two. While No.14 viewed a method primarily as a product and secondarily as a process, No.56 viewed a method primarily as a process and secondarily as a product. She explained this point well when asked to explain the mechanism of the passive flow:

> When you run, like, p(5, 1), it stopped in this position after result [pointing to p(5, 0)]. It never returned anything because it didn’t know what p(5, 0) was. So once you find out what p(5, 0) is, it starts where it needed to diverge off, and then it finishes. (Participant No.56 evaluating the number-prefix method)

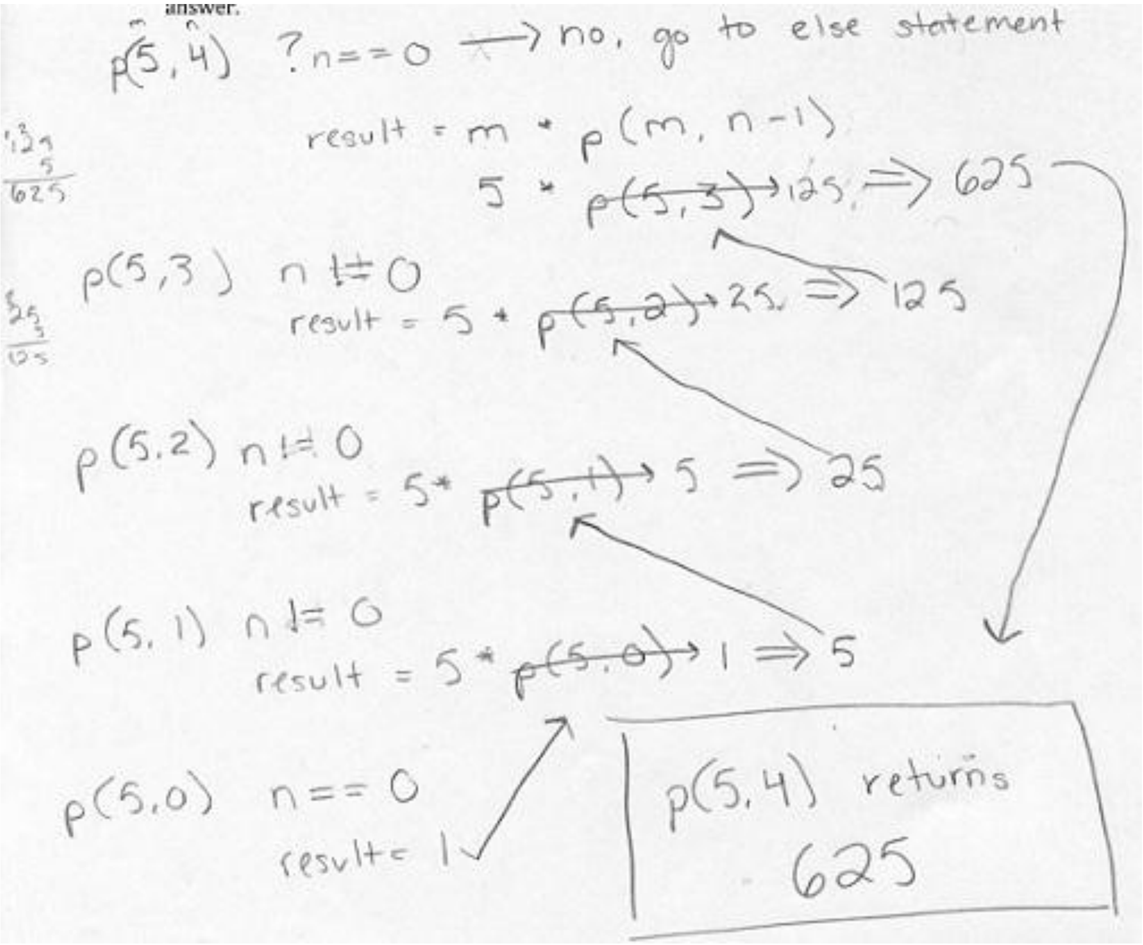


Figure 20. Participant No.56's evaluation trace for the number-prefix method.

The three mental transaction units processing the statement m*p(m, n-1) in the first three invocations were critical to generating the copies model. *Figure 21* shows a schematic diagram of the last unit 5*p(5, 0). At the beginning of this unit, she focused on the note "5*p(5, 0)" that she wrote after evaluating the IF statement in invocation p(5, 1). This note directly became the sensory input, which activated and instantiated the computer-as-agent p-prim. The patient component "p(5, 0)" unconditionally activated the process-to-product schema, which means the participant viewed the recursive call as a series of actions that produces a value. This process-to-product schema could not fill p-prim's effect slot thus required actions to diverge. On one hand, she stored the pending p-prim operation in her short-term memory. On the other hand, she embarked on evaluation of invocation p(5, 0). Once she got the value of p(5, 0), application of the process-to-product schema was complete, which allowed the pending p-prim operation to reactivate and reinstantiate with updated information. The p-prim operation completed with a simple multiplication. Then she replaced the note "p(5, 0)" with "1" and wrote out the result "=5".

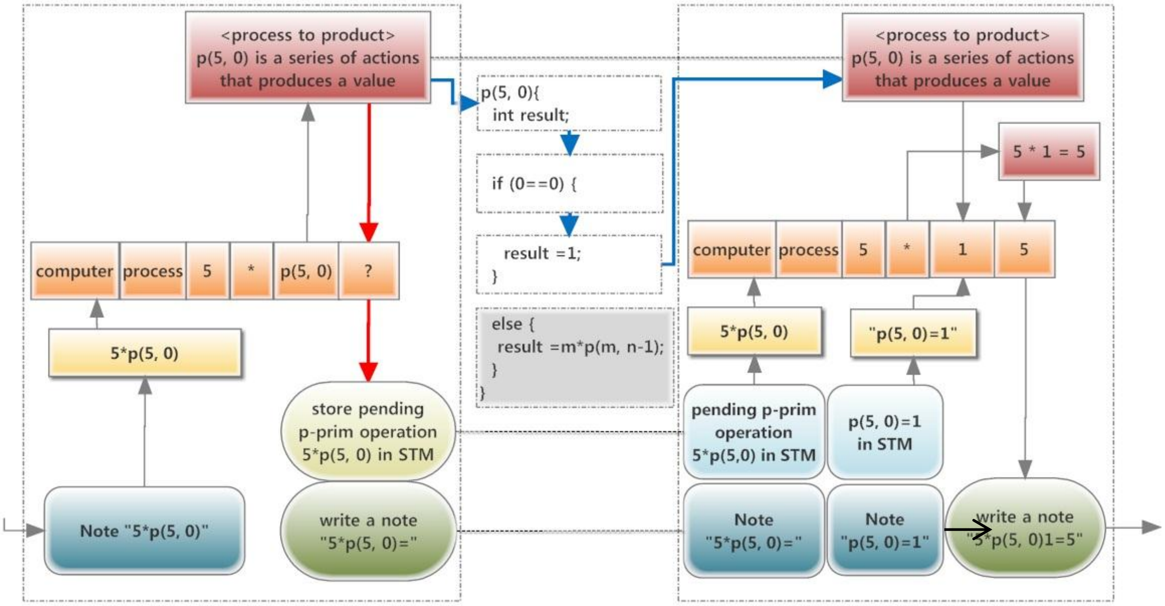


Figure 21. Schematic diagram of participant No.56's mental transaction unit of "5*p(5, 0)".

Distribution of Coordination Modes

As shown in Table 30, the three major coordination modes are evenly distributed among the Study 2 sample. The stable and unstable modes are also evenly distributed within the major categories.

Table 30
Distribution of coordination modes

Coordination modes	Frequency (proportion)
Unconditional coordination	10 (36%)
- Stable	- 6 (21%)
- Unstable	- 4 (14%)
Conditional coordination	9 (32%)
- Stable	- 4 (14%)
- Unstable	- 5 (18%)
Incoordination	9 (32%)
Total	28

Integrated Analyses

Cross-validation

The purpose of cross-validation is to examine the assumptions made in Study 1 regarding task design. In Study 1, situational constraint was estimated using two task dimensions (i.e., parameter type and call structure) that were previously shown to influence participants' performance in different ways. The trace data themselves cannot fully reveal the actual mechanism underlying situational constraint. The interview data revealed nuanced processes through which interaction between the participants and the tasks generated situational constraint.

Among the 28 interview participants, 10 performed inconsistently across the tasks. Of these, 6 participants used the process-related schemas in the prefix tasks but the entrance schema in the tail tasks (see Table 31).

Table 31
Six interview participants used different schemas of sub-invocations for prefix tasks and tail tasks

Participant	Tasks			
	Number-prefix	Number-tail	list-prefix	List-tail
No.14	Product-by-process	Entrance	Product-by-process	Entrance
No.54	Product-by-process	Entrance	Product-by-process	Entrance
No.40	Process-to-product	Entrance	Process-to-product	Entrance
No.24	Process-to-product	Entrance	Process-to-product	Entrance
No.39	Product-by-process	Entrance	-	-
No.33	-	-	Process-to-product	Entrance

Participants' reports of their evaluation processes clearly reflect how the prefix call structure and tail call structure differentially influence their interpretations of the sub-invocations. In the prefix methods, sub-invocations are components of recursive statements. Participants' interpretations of the recursive statements appear to directly influence how they interpret sub-invocations. After specifying a recursive statement with variable values, they often immediately note the need to obtain the value for the sub-invocation component. Taking participant No.14 as an example, her interpretation of the recursive statement "result = m * p (m, n-1)" in the number-prefix method prescribes that the sub-invocation "p (m, n-1)" is a numerical value, an attribute that favors the product-related schemas:

> …the result equals 5 times p of 5 comma 3, so then I was like ok I need to figure out what 5 times 3 is, so I went back through it……the result is 5 time p of 5 comma 2, so I still don't know what p of 5 comma 2 was, so I gotta figure that out……I did p of 5 comma 2 equals 5 times p of 5 comma 1, I still don't know what p of 5 comma 1 was……(participant No.14 evaluating the number-prefix method)

However, when she evaluated the number-tail method, the sub-invocations consistently activated the entrance schema instead:

> …I started at g of 10 comma 3……then you do g of 10 comma 2…so I went through back at 10 times 2 at the top……so then I would do g of 10 1, so go back to the top……so I had g of 10 comma 0, and I went back to the top, 10 times 0 is 0, and then, 0, this if statement does not apply anymore because n is equal to 0, and so I was like, I don't know what to do now, because I thought that would be the end of the method. (participant No.14 evaluating the number-tail method)

In contrast, of the 10 interview participants who performed inconsistently across the tasks, only one of them (participant No.39) used different schemas for the number tasks and the list tasks. When asked why she evaluated the number-prefix method and the list-prefix differently, participant No.39 thought for a while and then vaguely noted that it was unnecessary to return results to calling invocations in the list-prefix method as it was necessary to do so in the number-prefix method:

> …hmmm, because, because in this one (number-prefix method)…you are taking m and you are multiplying it by the recursion, uh… you need that value in order to multiply together. And this one (list-prefix method), you have, you can get i, you can get i plus 1, so you run through the whole thing, don't have to kind of get the previous values to getting this one (participant No.39 explaining difference in her performances on the number-prefix task and the list-prefix task).

It appears that she perceived the logical-AND operator in the list-prefix method differently from the multiplication operator in the number-prefix method. The source of situational constraint appears to be the participant's differential familiarity with the number domain and the Boolean domain, not the differential familiarity with the number domain and the list domain.

Following this clue, additional analysis is focused on participants' interpretations of the logical-AND operator in the list-prefix method. As the results reveal, participants' interpretations of the logical-AND operator had substantial impacts on their interpretations of sub-invocations. Nine participants[6] treated the logical-AND operator as a conjunction sign that connect two elements, two participants[7] treated it as a part of the output, and one participant[8] treated it as a punctuation sign without a substantive function. All these mistakes consequentially eliminated the need to return results to calling invocations. Three of these participants could have performed well if they understood the logical-AND operator, given that they demonstrated the copies model in evaluation of the number-prefix method.

Supposition 4

Supposition 4: Stability of the copies model is associated with coordination mode. The more coordinated the operation of the computer-as-agent p-prim, the more stable the copies model will be.

As shown in Table 32, the copies model is highly stable when p-prim operates in the unconditional coordination mode. Its stability is minimal when p-prim operates in the incoordination mode. When p-prim operates in three other modes, the copies model stability varies from absent to moderate. The result of Fisher's exact test is p < .001, indicating that coordination modes are significantly associated with stability of the copies model. A

[6] Participant No.7, No. 17, No.24, No.29, No.35, No.42, No.47, No.50, No.60
[7] Participant No.19 and No.34
[8] Participant No.59

Spearman's Rank Order correlation indicates a strong, positive correlation between participants' coordination modes and the stability of their copies model ($r_s = .800$, $p < .001$).

Table 32
Distribution of coordination modes among various levels of copies model stability

Coordination modes	Stability of the copies model					
	Absence	Low	Moderate low	Moderate	Moderate high	High
Stable unconditional					3	3
Unstable unconditional	1	1	1	1		
Stable conditional		1	1	2		
Unstable conditional		3	2			
Incoordination	5	4				
Total	6	9	4	3	3	3

CHAPTER 5

DISCUSSION

This study investigates the sources of difficulty that beginning computer science students have in understanding recursion. Specifically, it attempts to explain students' inconsistent performance across tasks targeting the recursion concept. Previous studies conceptualized coarse-grained mental models underlying various types of performance but failed to explain why these mental models manifest themselves inconsistently across similar tasks. Based on the knowledge-in-pieces perspective adopted in the current study, it was hypothesized that participants rely on the computer-as-agent p-prim to interpret recursive functions, and different task features differentially constrain the influences of this p-prim on performance. This subtle mechanism gives rise to the inconsistent performance observed across tasks that target the same concept.

To evaluate this general hypothesis, participants were asked to complete four tasks representing varying levels of constraint and interviewed to report and explain their thought processes. It was expected that more participants would demonstrate the copies models of recursion in the high-constraint tasks than in the low-constraint tasks, and for each individual participant, the copies model would cluster around tasks with relatively high constraint. Also, participants' interpretations of the recursive functions were expected to demonstrate characteristics associated with p-prim-generated interpretations.

Hypothesis 1

Hypothesis 1: Participants are more likely to exhibit the copies model when evaluating a recursive function with a prefix call structure than evaluating a recursive function with a tail call structure.

Number-Prefix versus Number-Tail

Results from the untriangulated trace data show that the proportion of participants demonstrating the copies model in the number-prefix task is significantly higher than the proportion of participants demonstrating the copies model in the list-prefix task. Results from the triangulated trace data show the same. As defined earlier (see Chapter 3), the number-prefix task and the number-tail task induce high and medium situational constraints, respectively. The higher the situational constraint, the more likely the computer-as-agent p-prim is to operate in appropriate ways. Thus, these results suggest that participants' inconsistent performance is attributable to varying operations of the computer-as-agent p-prim in different task situations[9].

[9] These results might be alternatively explained by effect of learning transfer (Singley & Anderson, 1989). The number-prefix task may represent a near-transfer task because the participants were predominantly exposed to similar examples prior to the quiz. Three of the

Validation using the interview data confirms that the call structure dimension successfully induces different levels of situational constraint. Analysis of the interview transcripts show that the presence of the prefix operation in the number-prefix task facilitates participants' consideration of the recursive call as a value produced by an invocation. In terms of knowledge coordination, activation of the multiplication schema raises the cuing priority of the product-by-process schema (a special case of the process schema). In contrast, the number-tail task does not constrain participants to properly interpret the recursive call. As a result, participants take alternative views of the recursive call (e.g., entrance schema), which

four examples introduced in the lecture had number parameters and prefix call structure, and four of the five tasks in the homework assignment also had this configuration. In contrast, the number-tail task may represent a far-transfer task because the participants had limited experience with recursive methods that involve tail call structure and printing statement. The teaching assistant reviewed two examples that covering these features right before the quiz, but such a quick exposure to the tail call structure cannot match the substantial experience with the prefix call structure.

This alternative explanation is not consistent with the observed performance on the quiz. Participants may demonstrate near-transfer by employing the same evaluation strategy on the quiz as they practiced in the homework assignment (Singley & Anderson, 1989). However, the assignment worksheet framed a bottom-up evaluation strategy in order to help participants understand the relationships among invocations. With this strategy, participants first evaluate the base case and then accumulate the results through the second last invocation to the original invocation. Only three participants employed this bottom-up strategy in the quiz. Other participants' traces all started with the given invocation and presented a typical sequence: active flow, base case, and passive flow if any. Thus, it is reasonable to believe that most of the participants consciously applied their knowledge of recursion rather than simply repeated their prior performance.

lead to construction of various non-copies models.

List-Prefix versus List-Tail

Results from the original trace data show that the proportion of participants demonstrating the copies model in the list-prefix task is not significantly different from the proportion of participants demonstrating the copies model in the list-tail task. Results from the triangulated trace data, however, do show the expected difference. As defined earlier (see Chapter 3), the list-prefix task and the list-tail task induce medium and low situational constraints respectively. Thus, the results partially suggest that participants' inconsistent performance is attributable to varying operations of the computer-as-agent p-prim in different task situations.

This discrepancy between the untriangulated trace data and the triangulated trace data is attributable to the recategorization of mental models based on the interview data. The mental models exhibited in the trace data do not necessarily reflect the quality of participants' knowledge because participants likely "satisfice" to fulfill the requirements of the assessment task (i.e., quiz) rather than to fully display their knowledge (Simon, 1956, p.129). In the case of this study, the list-prefix task permits two types of correct answer that may conceal one's understanding of recursion. First, participants may determine the final output within the original invocation, and eliminate the need to elaborate the recursive process (i.e., shortcut model). Second, participants may demonstrate the active flow and then combine results from all invocations without specifying the sequence of processing (i.e., active [combine-all-after base case] model). In the interviews, six of these participants showed the potential to demonstrate the copies model if they were required to fully elaborate the recursive process[10]. Recategorizing these participants' mental models substantially changes the result.

The results from the triangulated trace data are consistent with the potential effect of call structure emerged from previously reported data (Sanders et al., 2006). The research design in this study improves the credibility of this potential effect. Sanders and colleagues collected data from four cohorts of participants who each completed different tasks. This multi-cohort design introduced confounding factors due to varying cohort characteristics. The one-cohort design in the current study effectively eliminated these potential confounding factors. Also, the tasks used in the Sanders study had mixed and complex call structures (i.e., head-and-prefix call and prioritized prefix-and-suffix call), which made it difficult to isolate effects from each feature. In the current study, task features are simplified and controlled to permit more direct links between specified task features and participants' performance.

Hypothesis 2

Hypothesis 2: Participants are more likely to exhibit the copies model when evaluating a

[10] It is assumed that participants' understanding of recursion remained the same between the quiz and the interview for three reasons. First, participants generally demonstrated high levels of recall during interviews, which took place 2-10 days after they took the quiz. Second, the instructors did not provide additional instruction on recursion or review the quiz during the time period between the quiz and the interview. Third, participants were specifically asked to report their understanding during the quiz day.

recursive function with a number parameter than evaluating a recursive function with a list parameter.

Number-Prefix versus List-Prefix

Results from the untriangulated trace data show that the proportion of participants demonstrating the copies model in the number-prefix task is significantly higher than the proportion of participants demonstrating the copies model in the list-prefix task. Results from the triangulated trace data show the same. As defined earlier (see Chapter 3), the number-prefix task and list-prefix task induce high and medium situational constraint respectively. Thus, these results suggest that participants' inconsistent performance is attributable to varying operations of the computer-as-agent p-prim in different task situations.

However, validation using the interview data problematizes the parameter type dimension as a valid inducer of situational constraint. Analysis of the interview transcripts show that mistakes on the list operations have no impact on how participants construct mental models of recursion because they are not related to the recursive call. Instead, mistakes on the logical-AND operator (i.e., double ampersand "&&") substantially influence participants' performance. Because the logical-AND operator connects a Boolean value and the recursive call, a correct understanding of the logical-AND operator facilitates participants to recognize that the recursive call must return a Boolean value. In terms of knowledge coordination, activation of the proper logical-AND schema raises the cuing priority of the product-by-process schema. Several participants misunderstood the logical-AND operator as a punctuation or a connector between two elements. These misunderstandings remove the facilitation for correctly understanding the recursive call, leading to activation of other schemas that generate non-copies models.

Although parameter type failed to induce varying situational constraint in this study, it can do so if the list-prefix task is designed differently. In a previous study comparing participants' performance on a list function and a number function (Segal, 1995), both the list operations and the numerical operations were designed to be attached to the recursive call. Participants showed better performance on the number function than the list function. Segal speculated that participants' ability to view unprocessed operations in their entirety promotes a sound evaluation strategy in which processing of the operations is delayed until all invocations are fully instantiated. This ability is more developed in the number domain than in the list domain, so participants performed better on the number function than on the list function. From the perspective of the current study, this speculation is still plausible because it shows how participants' prior knowledge may render the same task high-constraint or low-constraint.

Contrasting the present study with Segal (1995) leads to a post hoc explanation for the present results. That is, situational constraint varies with participants' familiarity with the operations attached to the recursive call. These operations impose high situational constraint when participants have a good understanding of them, and they impose low situational constraint when participants have a poor understanding of them. In the case of the two tasks in this study, participants are more likely to be familiar with the multiplication operation in the number-prefix task than the logical-AND operator in the list-prefix task. Therefore, the number-prefix task in general imposes higher situational constraint than the list-prefix task does.

Number-Tail versus List-Tail

Results from the untriangulated trace data shows that the proportion of participants demonstrating the copies model in the number-tail task is not significantly different from the proportion of participants demonstrating the copies model in the list-tail task. Results from the triangulated trace data do not show any difference either. As defined earlier (see Chapter 3), the number-tail task and list-tail task induce medium and low situational constraints respectively. Thus, these results do not suggest that participants' inconsistent performance is attributable to varying operations of the computer-as-agent p-prim in different task situations.

These unexpected results may be explained by the aforementioned issue of parameter type failing to impose varying levels of situational constraint. In both of these tasks, there are no operations attached to the recursive call. Participants' misunderstandings on the unattached operations, either numerical operations or list operations, would not affect how they evaluate the recursive call. Thus, participants' differential familiarity with the two domains does not influence their performance.

Hypothesis 3

Hypothesis 3: If participants exhibit the copies model in some but not all task situations that represent four possible combinations of call structure and parameter type, their successful performance will be clustered around tasks with higher situational constraints. Performance patterns representing such trend should occur more frequently than those contradicting it.

Results from the untriangulated trace data show that the proportion of hypothesis-consistent performance patterns significantly exceeds the proportion expected based on a random distribution. Results from the triangulated trace data show the same. These results indicate that for the participants with a tentative understanding of recursion, successful performance is a function of situational constraint induced by a task[11].

These results importantly complement the results for hypothesis 1 and hypothesis 2. The first two sets of results show participants' inconsistent performance at the group level. However, because the set of participants demonstrating the copies model in the high-constraint task overlaps with rather than encompasses the set of participants demonstrating the copies model in the low-constraint task, it would be unsound to infer inconsistent performance at the individual level. Therefore, this set of results is necessary to demonstrate inconsistent performance at the individual level.

In sum, these results largely support the general hypothesis that participants' inconsistent performance is attributable to varying operations of the computer-as-agent p-prim in different task situations.

[11] Considering the aforementioned issue of parameter type failing to induce a varying situational constraint, cases only showing differential performance by parameter type should not be considered. Fortunately, only one participant from the untriangulated trace data shows such pattern combinations. Removing this case from the analysis would not change the results.

Supposition 1

Supposition 1: Interpretations generated by the computer-as-agent p-prim are characteristic of a sense of intuitive obviousness and satisfaction.

The interview data is consistent with the expected self-explanatory characteristic of the interpretations generated by computer-as-agent p-prim. A sense of certainty and satisfaction, as shown in participants' tones and linguistic features, accompanies these interpretations, regardless of whether they are productive or unproductive. These interpretations are robust enough to be articulated, even when participants interpret a function as being useless (i.e. attaining a null output). Contrast between these interpretations and standard interpretations generate puzzlement and confusion as well as an expressed need for explanations of the unanticipated outcomes.

Supposition 2

Supposition 2: Interpretations generated by the computer-as-agent p-prim serve participants well in many cases rather than causing problems all the time.

Interpretations of non-recursive statements and original invocation are mostly productive, although these interpretations have limited contribution to construction of the copies model of recursion. Interpretations of the sub-invocations and recursive statements can be productive or not depending on the particular contextual configurations shaped by task features. Overall, the interpretations generated by the computer-as-agent p-prim are functional.

Supposition 3

Supposition 3: Interpretations generated by the computer-as-agent p-prim are sensitive to local context, thus they are likely to change in response to change in local context.

Inspection across mental transaction units shows that interpretations of the recursive call are inconsistent across tasks. Substantially different contextual configurations may activate different schemas through the computer-as-agent p-prim, resulting in differing interpretations. The slightly different contextual configurations may activate the same schema but charge it with different levels of cuing reliability, leading to inconsistent performance in a single task.

In sum, participants' interpretations of recursive functions show the main characteristics of p-prim based interpretations (diSessa, 1993): self-explanatory, functional, and context-sensitive.

Supposition 4

Hypothesis 4: Stability of the copies model is associated with coordination mode. The more coordinated the p-prim operation, the more stable the copies model

Results indicate that the computer-as-agent p-prim's coordination modes are associated with the stability of the copies model. When the computer-as-agent p-prim unconditionally coordinates with the process schema, the copies model is highly stable. When the coordination occurs conditionally, the stability of the copies model is fair. When no coordination exist whatsoever, the copies model is extremely unstable. These findings suggest that participants' understanding of recursion can be described in terms of coordination among knowledge elements, because such descriptions successfully account for the unexplained variability in participants' performance.

This finding is new to the literature on learning recursion. Many researchers (e.g., Götschi, 2003; Mirolo, 2010; Scholtz & Sanders, 2010) described participants' understanding of recursion in terms of mental models and associated misconceptions. However, no research describes it in terms of the quality of knowledge system or the coordination of a computer-as-agent p-prim with essential schemas, consistent with the knowledge-in-pieces perspective.

Conclusions

These two studies together address the following two questions:

1. Do beginning CS students demonstrate reliance on identifiable p-prims when trying to understand and apply recursion?
2. If they do, what are the structures, relevant circumstances, functions, and effects of these p-prims as they impact learning and performance?

The combined results indicate that beginning CS students rely on the computer-as-agent p-prim when trying to understand and apply recursion. Study 1 demonstrates that participants' inconsistent performance on recursive function evaluation can be attributed to varying operations of a computer-as-agent p-prim in tasks with different features. Findings from Study 2 suggest that participants' interpretations generated by the computer-as-agent p-prim are self-explanatory, functional, and context-sensitive, which are the hallmarks of p-prim-based interpretations.

On the basis of these results, a preliminary knowledge-in-pieces-based model of recursive function evaluation can be developed that grounds the functioning of the identified p-prim. The following section describes this model, emphasizing the computer-as-agent p-prim.

A Model of Recursive Function Evaluation

Built on the general knowledge-in-pieces framework (diSessa, 1993), the computer-as-agent p-prim and programming schemas comprise a cognitive model of recursive function evaluation. These knowledge elements constitute a knowledge activation network with structured priorities. A person's ability to correctly evaluate recursive functions is a function of the mode of coordination between the computer-as-agent p-prim and the process-related schemas of method invocation.

Computer-as-Agent P-prim Properties

Schematization: The computer-as-agent p-prim has four components structured in an agentive format: computer (agent) processes (action) instructions (patient) to generate results (effect).

Circumstances: original method invocation, sub-invocation, recursive statement, and non-recursive statement are the circumstances that activate the computer-as-agent p-prim.

Operation: When the computer-as-agent p-prim is activated, it fills up its slots with contents. In the case of program evaluation, the patient slot is occupied first by the elements of the given function. Then, feedback from programming schemas occupies the effect slot.

Function: The computer-as-agent p-prim functions as an interpretive framework for program elements and generates layers of control over the program evaluation process.

Programming Schemas

Programming schemas are action-oriented mental representations of the elements in a program. They are activated by their corresponding circumstances—a certain element in a certain context. Then, they generate specific actions upon the program elements. The same program element may have multiple alternative schemas, which are sensitive to slightly different contexts around the program element. In the present study, schemas of method invocation (original invocation or sub-invocation) and schemas of logical-AND operator appear to affect whether participants demonstrate the copies model of recursion. The copies model of recursion relies on coordination between the computer-as-agent p-prim and the process-related schemas of method invocation, and the proper logical-AND schema helps activate these process-related schemas.

Theoretical Cognitive Mechanism

The sub-invocation circumstance first activates the computer-as-agent p-prim. The activated computer-as-agent p-prim specifies the context, which further activates the schemas of method invocation. The activated schema completes its operation and sends feedback to the computer-as-agent p-prim. The computer-as-agent p-prim completes its operation once all its slots are filled. Coordination refers to a scenario in which the computer-as-agent p-prim and the schema of method invocation are active at the same time to accomplish a goal.

In a developing knowledge system, the sub-invocation circumstance may activate one of the schemas of method invocation because they all have a similar cuing priority to the circumstance. In such case, the computer-as-agent p-prim plays an important but problematic role. The computer-as-agent p-prim coordinates with the non-process-related schemas because they usually entail only a few actions and immediately send feedback to the active computer-as-agent p-prim. In contrast, the process-related schemas entail many more actions and take longer to send feedback. Thus, the active computer-as-agent p-prim shapes the context in a way that raises the cuing priority of the non-process-related schemas.

However, other active knowledge elements in the context can constrain the dominant influence of the computer-as-agent p-prim over which schemas of method invocation will be activated. For instance, when a recursive call has a prefix multiplication operation, the nature of this operation implies that the sub-invocation returns a number. The active attribute schema raises the cuing priority of the product-by-process or process-to-product schemas of method invocation which are compatible with prescribed attribute. Meanwhile, it lowers the cuing priority of the non-process-related schemas that are preferred by the computer-as-agent p-prim. Thus, this active attribute schema weakens the unproductive influence of the computer-as-agent p-prim.

In a developed knowledge system, the process-related schemas have developed high cuing priorities to the sub-invocation circumstance. The active computer-as-agent p-prim still shapes the context, but its influence is limited compared to the established high cuing priority of the process-related schemas.

Three Modes of Coordination

In order to construct a copies model of recursion, it is necessary for the computer-as-agent p-prim to coordinate with the process-related schemas. Three coordination modes can be observed when knowledge systems at different developmental stages interact with tasks with various features.

In the incoordination mode, the computer-as-agent p-prim does not coordinate with the process-related schemas in any of the tasks. Instead, it coordinates with the non-process-related schemas (e.g., variable updater, etc.) cued by some features of the sub-invocation circumstance (e.g., variable value decrementing by 1). In the knowledge system, either the process-related schemas have not emerged yet, or their cuing priorities to the sub-invocation circumstance are so low that even constraining knowledge elements could not help activating them.

In the conditional coordination mode, the computer-as-agent p-prim coordinates with the process-related schemas in some but not all the tasks. In the knowledge system, the process-related schemas have developed competitive cuing priorities relative to the non-process-related schemas. However, when no constraining knowledge element is present, the computer-as-agent p-prim favors the non-process-related schemas. So for the coordination to occur, some constraining knowledge elements must be active to weaken the unproductive influence of the computer-as-agent p-prim. These constraining knowledge elements can be activated by some tasks with certain features but not the others.

In the unconditional coordination mode, the computer-as-agent p-prim coordinates with the process-related schema in all the tasks. In the knowledge system, the process-related schemas have developed high cuing priorities to the sub-invocation circumstance. With the established cuing priority in place, the unproductive influence of the computer-as-agent p-prim is negligible. Also, the presence of constraining knowledge elements is only reinforcing but not essential to the activation of the process-related schemas.

Implications for Theory

As discussed in Chapter 2, inspired by the mental model theory (Gentner & Stevens, 1983; Johnson-Laird, 1983), many researchers (e.g., Götschi, Sanders, & Galpin, 2003; Kahney, 1983) successfully identified a variety of mental models of recursion among novice programmers. Although they conceptualized the mental models of recursion as integral knowledge structures that underlie participants' interpretations of recursive functions, this particular conceptualization does not lead to a plausible explanation for the inconsistent manifestation of these mental models across similar tasks.

However, in Johnson-Laird's (1983) mental model theory of reasoning, the mental models of syllogism are configured with only a few parts and cannot be further decomposed to meaningful units. Collins and Gentner (1987) studied mental models of relatively complex phenomena, and they considered these mental models as constructed by combining several “component models” (p.254). These original accounts point to the potential benefit of identifying the smallest knowledge structures that constitute large and complex mental models.

The present study does exactly this; it focuses its investigation on finer-grained knowledge structures. The mental model of recursion is reconceptualized as a descriptive construct for patterns of performance, as opposed to an explanatory integral knowledge structure as previously conceptualized in the literature. Accordingly, in this study, misconceptions are conceptualized as facets of maladaptive mental models rather than casual knowledge structures. As a result, the inconsistency in mental model manifestation is a phenomenon to be explained by behaviors of the finer-grained knowledge structures.

As the mental model theory lacks an elaborated account of these elemental knowledge structures, the knowledge-in-pieces theory duly serves as an effective framework. The proposed model explains participants' inconsistent performance on recursive function evaluation that previous studies were not able to explain. This outcome reflects the importance of breaking a mental model into components and describing the relationships among them in order to sufficiently explain the observations at the mental model's level.

Situational Constraint

The three hypotheses posed in this study require the concept of situational constraint. Although the term is new to the conceptual change literature, the concept itself is present in a variety of previous studies of human knowledge systems (Frank, 2010; Parnafes, 2005; Thaden-Koch, Dufresne, & Mestre, 2006; diSessa, 1993). For example, Thaden-Koch et al. (2006) found that in a certain task setting but not others, physics participants' reliance on their novice knowledge in physics consistently steered them away from useful and directly observable information in the given task environment, whereas their non-physics counterparts directly perceived these information and use them to generate sound interpretations. Thus, a person's behaviors are neither determined by the properties of the person nor by the properties of the environment alone. Rather, the behaviors are constrained by some properties of the entire person-environment system.

Although the concept of situational constraint is frequently used to provide post hoc explanations, it has not been explicitly clarified and systemically utilized to generate testable hypotheses. It is probably because the methodology of conceptual change studies has been predominately inductive, which does not require a priori explanations. However, for the knowledge-in-pieces theory to advance, investigation must move from only using inductive approaches to using deduction or mixing the two approaches. The concept of situational constraint is important to making this move, because it is a tool to predict consequences of task design for certain population.

To clearly define situational constraint, a definition of *situation* must be in place first. In the present study, situation is defined as the state of a nondecomposable person-environment system. The grain size of a situation can be as large as participant-taking-a-quiz-in-half-an-hour or as small as participant-evaluating-a-recursive-call-in-a-few-seconds. Situational constraint is thus a property of the situation pertaining to the operations of certain knowledge elements. In this study, the knowledge element of interest is the computer-as-agent p-prim.

As the grain sizes of situations vary, situational constraint may refer to how a transient context specified by active knowledge elements constrains the influence of the computer-as-agent p-prim over further knowledge activation. Alternatively, situational constraint may also refer to the overall constraint that a coarse-grained situation (e.g., participant-evaluating-a-recursive-function) imposes on the computer-as-agent p-prim. This overall constraint can be estimated by aggregating the constraints imposed by the fine-grained situations. In this study, the task-level situational constraint is used to make assumptions underlying task design for hypotheses testing.

It is usually difficult to predict a series of situations that a person-environment system would move through. Indeed, such predictions require strict control of the task environment and accurate knowledge of the person's task-relevant abilities. However, it is not impossible if two conditions are in place: (1) a rich literature base informative about potentially important task properties and (2) adequate information about the sample's characteristics. The present study is fortunate to have both. The tasks are designed to vary only on two important dimensions which were identified from the literature, and participants' task-relevant abilities are determined based on the information about the specific class. Then, the situational constraints of the tasks are estimated by simulating participants' task performances using the properties of both the task environment and the participants' knowledge system.

Despite the technical challenge just described, the concept of situational constraint should lend itself to being a productive ingredient in investigations of complex knowledge system.

Extending Knowledge-in-Pieces Perspective to the Domain of Computing

In this study, the agentive causality meta-p-prim in intuitive physics (diSessa, 1993) also manifests in the domain of computing. This finding suggests that knowledge systems of different domains do share nontrivial elements. Indeed, domain boundaries are probably not an intrinsic property of intuitive knowledge systems, and the same element may be used for reasoning in multiple domains (Russ, Sherin, & Drive, 2008).

It may be counterintuitive for p-prims to exist in an intuitive computing knowledge system. Certainly, computing activity seems to be highly symbolic, not relying on bodily experiences like physical activities do. However, computing is essentially a form of problem solving, which is an everyday task at a multiple of scales. Although every problem is unique in some way, there are patterns shared in all problem solving activities (Newell & Simon, 1972). Engagement in problem solving generates a rich experiential base for the domain of computing.

Some abstractions from human problem solving turn out to be inappropriate for computing. For instance, computer programs written by younger participants frequently contain errors generated by inappropriate assumptions that computer programs are capable of interpreting the programmer's intentions (Pea, 1986). Such preconceptions are probably developed from collaborative problem solving activities in which agents are human interpreters.

Direct interaction with computing devices is another important source for intuitive computing knowledge. As human society becomes increasingly reliant on information technology, children are engaged in computer-mediated activities at increasingly earlier ages and with increasing frequency (Newburger, 2009). As a result, children develop conceptions of computers at a younger age than before (Oleson, Sims, Chin, Lum, & Sinatra, 2010). Primitive knowledge structures may emerge from children's direct experiences with computers at early developmental stages. Such first-handedly originated p-prims might be particularly problematic because they are rooted in bodily experiences just like the p-prims of mechanism are.

Implications for Practice

This knowledge-in-pieces-based explanation has implications for design of diagnostic assessment of recursion. Traditionally, diagnostic assessments are designed to identify maladaptive mental models and associated misconceptions, but the inconsistency of these mental structures limits the usefulness of assessment (Götschi, 2003). In this study, mental models and misconceptions are redefined as manifestations of the underlying knowledge system. Within this framework, the goal of diagnostic assessment should be refocused on the quality of the underlying knowledge system. Test items should be designed to reflect various situational constraints. High-constraint tasks would not differentiate participants because they provide excessive assistance for participants to successfully accomplish the task. Low-constraint tasks would be ideal for identifying participants with high quality knowledge system. Medium-constraint tasks would be appropriate to examine specific aspects of participants' knowledge system.

This study also has implications for the optimal sequence for teaching programming concepts. Researchers disagree about when to teach recursion. Some suggest teaching recursion as an advanced topic and place it after looping construct (e.g., Kessler & Anderson, 1986; Wiedenbeck, 1988), while others suggest teaching recursion prior to looping construct to prevent confusion (e.g., Levenick, 1990; Turbak et al., 1999). These arguments are framed around the belief that maladaptive mental models and associated misconceptions are attributable to prior knowledge of concepts resembling recursion. This study alternatively concludes that maladaptive mental models are attributable to participants' own intuition.

However, this conclusion does not exclude the possibility that learning certain programming construct in prior may strengthen or weaken certain knowledge activation patterns. For instance, when evaluating simple non-recursive functions, it is sufficient to use the entrance schema of method invocation because there is no chance to leave the invocation during evaluation until reaching the end of the function. Repeated experiences with such functions may strengthen the coordination between the computer-as-agent p-prim and the entrance schema. The looping construct may be a special case of such experiences because repeatedly applying the entrance schema to the loop section is sufficient to generate correct evaluation.

This study also has implications for selection of appropriate examples to introduce recursion. Typically, CS textbooks and instructors introduce recursion using mathematical functions such as factorials or Fibonacci numbers. These functions are sound choices due to their simplicity and practical utility, but they cover only a very limited range of task features (i.e. number parameter and fixated call structure). These features do not require unconditional coordination due to their high situational constraint. Without actually practicing with low-constraint tasks, participants' knowledge system will remain unable to unconditionally coordinate appropriate knowledge elements. Thus, instruction and practice may start with low-constraint tasks such as evaluating methods with tail call structure. These types of methods may lack practical utility. Instructors may consider mixing them with those high-constraint tasks traditionally used and highlight the differences between the two. Such instructional practice is consistent with the established evidence that varied practice facilitates learner to extract invariant information among tasks, which is essential to transfer of learning

(Anderson, 1983).

Limitations

The researcher only recruited participants from one class at one university. Because of the high ranking of the university (national top 25), this sample of participants is likely to be more homogeneous and have stronger academic ability than the general population. This sample characteristic limits the extent to which the results and conclusions can be generalized. However, the study successfully captured a full range of performance patterns, suggesting reasonable variability within the sample.

The researcher herself was the only coder and interpreter in this study. She independently developed the coding schemes, evaluated trace data, and analyzed interview transcripts. Although safeguards were utilized to limit bias (see Chapter 3), future studies would benefit from independent, blind coding by multiple researchers.

Future Directions

Future research may extend application of knowledge-in-pieces theory to investigate how learning of recursion occurs in terms of how knowledge system evolves in reaction to environmental perturbations. This study has provided sketches of three types of coordination. Assuming all participants share the same learning trajectory, it appears that knowledge system evolves from incoordination mode, to conditional coordination mode, and then to unconditional coordination mode. However, it is unclear whether participants share the same trajectory or there are multiple trajectories. More importantly, it is critical to know what conditions cause these changes. These nuanced processes of learning can be captured using microgenetic research method (Siegler & Crowley, 1991).

Also, future research may extend application of the knowledge-in-pieces theory to investigate how participants solve recursive problems because a well-rounded understanding of recursion consists of understanding of both the machine domain (addressed in this study) and the problem domain. As participants' understandings of recursive function are often associated with their understandings of recursive problems (Mirolo, 2010), the findings of this study are likely to guide the new investigation. The knowledge elements and cognitive dynamics identified in this study likely play important roles in recursive problem solving. For instance, an unconditional coordination between the computer-as-agent p-prim and the process schema of method invocation is essentially the ability to encapsulate a process and make it a part of a larger process—an ability critical to formulating a solution that refers to itself.

Building upon the well elaborated knowledge-in-pieces-based model of how participants learn recursive function (the machine domain) and recursive problem (the problem domain), research can be further focused on evaluating various existing instructional practices. These practices include various static (Er, 1995; Haynes, 1995; Kruse, 1982; Murnane, 1991; Troy & Early, 1992; Wu, 1993) and dynamic (George, 2000; Kann, Lindeman, & Heller, 1997; Wilcocks & Sanders, 1994) representations of the recursive process. Hypotheses generated based on the knowledge-in-pieces-based model may capture more nuanced differences among the practices because this study shows that the knowledge-in-pieces-based model has greater explanatory power than previous models.

REFERENCES

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs* (2nd ed.). Cambridge, MA: MIT Press.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, *8*(2), 87–129.

Anderson, J. R., Pirolli, P. L., & Farrell, R. (1988). Learning to program recursive functions. In M. T. H. Chi, R. Glaser, & M. J. Farr (Eds.), *The nature of expertise* (pp. 153–184). Hillsdale, NJ: Lawrence Erlbaum Associates.

Augustine, N. R. (2005). *Rising above the gathering storm: Energizing and employing America for a brighter economic future*. Washington, D.C.: National Academies Press.

Bannon, L., & Bødker, S. (1991). Beyond the interface: Encountering artifacts in use. In J. M. Carroll (Ed.), *Designing interaction: Psychology at the human-computer interface* (pp. 227–253). New York: Cambridge University Press.

Beaubouef, T. (2005). Why the high attrition rate for computer science students: Some thoughts and observations. *ACM SIGCSE Bulletin*, *37*(2), 103–106.

Ben-Ari, M. (1998). Constructivism in computer science education. *ACM SIGCSE Bulletin*, *30*(1), 257–261.

Bhuiyan, S. H., Greer, J. E., & McCalla, G. I. (1990). Mental models of recursion and their use in the SCENT programming advisor. In S. Ramani, R. Chandrasekar, & K. S. R. Anjaneyulu (Eds.), *Knowledge based computer systems: International Conference KBCS'89, Bombay, India, December 11–13, 1989 proceedings* (pp. 135–144). Berlin: Springer-Verlag.

Bhuiyan, S. H., Greer, J. E., & McCalla, G. I. (1994). Supporting the learning of recursive problem solving. *Interactive Learning Environments*, *4*(2), 115–139.

Brown, N. J. S., Danish, J. A., DeLiema, D., Engle, R. A., Enyedy, N. D., Lee, V. R., & Parnafes, O. (2012). Representations, interlocutors, and their influences on apparent knowledgeability. *Paper presented at the Annual Meeting of the American Educational Research Association*. Vancouver, Canada.

Bureau of Labor Statistics. (n.d.). Employment projections—numerical and percent change, by detailed occupation. Retrieved May 17, 2011, from http://www.bls.gov/emp/ep_table_102.pdf

Chi, M. T. H. (1997). Quantifying qualitative analyses of verbal data: A practical guide. *Journal of the Learning Sciences*, *6*(3), 271–315.

Chi, M. T. H. (2008). Three types of conceptual change: Belief revision, mental model transformation, and categorical shift. In S. Vosniadou (Ed.), *International handbook of research on conceptual change* (pp. 61–82). New York: Routledge.

Chinn, D., Martin, K., & Spencer, C. (2007). Treisman workshops and student performance in CS. *ACM SIGCSE Bulletin*, *39*(1), 48–52.

Clement, J. (1982). Students' preconceptions in introductory mechanics. *American Journal of Physics*, *50*(1), 66–71.

Clement, J. (1993). Using bridging analogies and anchoring intuitions to deal with students' preconceptions in physics. *Journal of Research in Science Teaching*, *30*(10), 1241–1257.

Clement, J., & Steinberg, M. (2002). Step-wise evolution of mental models of electric circuits: A "learning-aloud" case study. *Journal of the Learning Sciences*, *11*(4), 389–452.

Cohoon, J. P. (2007). An introductory course format for promoting diversity and retention. *Proceedings of the 38th SIGCSE technical symposium on Computer Science Education* (pp. 395–399). New York: ACM.

Cohoon, J. P., & Davidson, J. (2006). *Java 5.0 Program Design*. New York: McGraw-Hill, Inc.

Cohoon, J. P., & Tychonievich, L. (2011). Analysis of a CS1 approach for attracting diverse and inexperienced students to computing majors. *Proceedings of the 42nd ACM technical symposium on Computer Science Education* (pp. 165–170). New York: ACM.

Collins, A., & Gentner, D. (1987). How people construct mental models. In D. Holland & N. Quinn (Eds.), *Cultural models in language and thought* (pp. 243–265). Cambridge, England: Cambridge University Press.

Computing Research Association. (2011). *Taulbee survey report 2009–2010*. Retrieved May 31st, 2012, from http://www.cra.org/resources/taulbee/

Conway, M., & Kahney, H. (1987). Transfer of learning in acquiring the concept of recursion. In J. Hallam & C. Mellish (Eds.), *Advances in artificial intelligence: Proceedings of the 1987 AISB conference, University of Edinburgh, 6–10 April 1987* (pp. 239–250). Chichester: Wiley.

Craik, K. J. W. (1943). *The nature of explanation*. Cambridge: Cambridge University Press.

Creswell, J. W., & Plano Clark, V. L. (2007). *Designing and conducting mixed methods research*. Thousand Oaks, CA: Sage Publications.

da Rosa, S. (2007). The learning of recursive algorithms from a psychogenetic perspective. *Proceedings of the 19th Annual Psychology of Programming Interest Group Workshop, Joensuu, Finland, 2007* (pp. 201–215).

Dale, N. B. (2006). Most difficult topics in CS1: Results of an online survey of educators. *ACM SIGCSE Bulletin*, *38*(2), 49–53.

diCheva, D., & Close, J. (1996). Mental models of recursion. *Journal of Educational Computing Research*, *14*(1), 1–23.

diSessa, A. A. (1983). Phenomenology and the evolution of intuition. In D. Gentner & A. Stevens (Eds.), *Mental models* (pp. 15–33). Hillsdale, NJ: Lawrence Erlbaum Associates.

diSessa, A. A. (1988). Knowledge in pieces. In G. Forman & P. Pufall (Eds.), *Constructivism in the Computer Age* (pp. 49–70). Hillsdale, NJ: Lawrence Erlbaum Associates.

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

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

diSessa, A. A., Greeno, J. G., & Michaels, S. (2012). Perspectives on the clinical interview as an interactive genre. *Paper presented at the Annual Meeting of the American Educational Research Association*. Vancouver, Canada.

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

Doyle, J. K., & Ford, D. N. (1998). Mental models concepts for system dynamics research. *System dynamics review*, *14*(1), 3–29.

Driver, R., & Easley, J. (1978). Pupils and paradigms: A review of literature related to concept development in adolescent science students. *Studies in Science Education*, *5*(1), 61–84.

du Boulay, D., & O'Shea, T. (1981). Teaching novices programming. In M. Coombs & J. Alty (Eds.), *Computing skills and the user interface*. London, UK: Academic Press.

Er, M. (1995). Process frame: A cognitive device for recursion comprehension. *Computers & Education*, *24*(1), 31–36.

Ford, G. (1984). An implementation-independent approach to teaching recursion. *ACM SIGCSE Bulletin*, *16*(1), 213–216.

Frank, B. W. (2010). Multiple conceptual coherences in the speed tutorial: Micro-processes of local stability. In K. Gomez, L. Lyons, & J. Radinsky (Eds.), *ICLS'10 Proceedings of the 9th International Conference of the Learning Sciences - Volume 1* (Vol. 1, pp. 873–880). Chicago, IL: International Society of the Learning Sciences.

Gentner, D., & Stevens, A. L. (1983). *Mental models*. Hillsdale, NJ: Lawrence Erlbaum Associates.

George, C. E. (2000a). EROSI—Visualising recursion and discovering new errors. *ACM SIGCSE Bulletin*, *32*(1), 305–309.

George, C. E. (2000b). Experiences with novices: The importance of graphical representations in supporting mental models. In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group* (pp. 33–44). Corigliano Calabro, Italy: Edizioni Memoria.

Ginat, D. (2004). Do senior CS students capitalize on recursion? *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE'04* (pp. 82–86). New York: ACM.

Ginat, D., & Shifroni, E. (1999). Teaching recursion in a procedural environment—how much should we emphasize the computing model? *ACM SIGCSE Bulletin*, *31*(1), 127–131.

Ginsburg, H. (1981). The clinical interview in psychological research on mathematical thinking: Aims, rationales, techniques. *For the Learning of Mathematics*, *1*(3), 4–11.

Given, L. M. (2008). *The Sage encyclopedia of qualitative research methods, volume 2*. Thousand Oaks, CA: Sage Publications.

Glaser, R., & Bassok, M. (1989). Learning theory and the study of instruction. *Annual Review of Psychology*, *40*(1), 631–666.

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L. C., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a Delphi process. *ACM SIGCSE Bulletin*, *40*(1), 256.

Goldschlager, L., & Lister, A. (1982). *Computer science: A modern introduction*. Englewood Cliffs, NJ: Prentice Hall.

Guzdial, M., & Soloway, E. M. (2002). Teaching the Nintendo generation to program. *Communications of the ACM*, *45*(4), 17–21.

Götschi, T. (2003). *Mental models of recursion*. (Unpublished Master's Thesis). University of Witwatersrand, Johannesburg, Gauteng, South Africa.

Götschi, T., Sanders, I., & Galpin, V. (2003). Mental models of recursion. *Proceedings of the 34th SIGCSE technical symposium on Computer science education - SIGCSE'03* (pp. 346–350). New York: ACM.

Haberman, B. (2004). How learning logic programming affects recursion comprehension. *Computer Science Education*, *14*(1), 37–53.

Haberman, B., & Averbuch, H. (2002). The case of base cases: Why are they so difficult to recognize? Student difficulties with recursion. *Proceedings of the 7th annual conference on innovation and technology in computer science education* (pp. 84–88). New York: ACM.

Halldén, O., Haglund, L., & Strömdahl, H. (2007). Conceptions and contexts: On the interpretation of interview and observational data. *Educational Psychologist*, *42*(1), 25–40.

Hammer, D., Elby, A., Scherr, R. E., & Redish, E. F. (2005). Resources, framing, and transfer. In J. P. Mestre (Ed.), *Transfer of learning from a modern multidisciplinary perspective* (pp. 89–120). Greenwich, CT: IAP.

Haynes, S. M. (1995). Explaining recursion to the unsophisticated. *ACM SIGCSE Bulletin*, *27*(3), 3–6.

Hewson, P. W., & Hewson, M. G. A. (1984). The role of conceptual conflict in conceptual change and the design of science instruction. *Instructional science*, *13*(1), 1–13.

Holyer, I. (1991). *Functional programming with Miranda*. London, UK: Pitman.

Howles, T. (2007). Preliminary results of a longitudinal study of computer science student trends, behaviors and preferences. *Journal of Computing Sciences in Colleges*, *22*(6), 18–27.

Howles, T. (2009). A study of attrition and the use of student learning communities in the computer science introductory programming sequence. *Computer Science Education*, *19*(1), 1–13.

Ifenthaler, D., Masduki, I., & Seel, N. M. (2011). The mystery of cognitive structure and how we can detect it: Tracking the development of cognitive structures over time. *Instructional Science*, *39*(1), 41–61.

Isbell, C. L., Xu, Y., Stein, L. A., Cutler, R., Forbes, J., Fraser, L., Impagliazzo, J., et al. (2010). (Re)defining computing curricula by (re)defining computing. *ACM SIGCSE Bulletin*, *41*(4), 195–207.

Johnson-Laird, P. N. (1980). Mental models in cognitive science. *Cognitive Science*, *4*(1), 71–115.

Johnson-Laird, P. N. (1983). *Mental models: Towards a cognitive science of language, inference, and consciousness*. Cambridge, England: Cambridge University Press.

Kahney, H. (1983). What do novice programmers know about recursion. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (pp. 235–239). New York: ACM.

Kahney, H., & Eisenstadt, M. (1982). Programmers’ mental models of their programming tasks: The interaction of real world knowledge and programming knowledge. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society* (pp. 143–145). Mahway, NJ: Lawrence Erlbaum Associates.

Kann, C., Lindeman, R. W., & Heller, R. (1997). Integrating algorithm animation into a learning environment. *Computers & Education*, *28*(4), 223–228.

Keane, M., Kahney, H., & Brayshaw, M. (1989). Simulating analogical mapping difficulties in recursion problems. *AISB89-Proceedings of the Seventh Conference of the Society for the Study of Artificial Intelligence and Simulation of Behaviour* (pp. 3–12). London: Morgan Kaufmann.

Kessler, C. M., & Anderson, J. R. (1986). Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, *2*(2), 135–166.

Klein, G. A., & Calderwood, R. (1996). *Investigations of naturalistic decision making and the recognition-primed decision model (research note 96–43)*. Yellow Springs, OH.

Kruse, R. L. (1982). On teaching recursion. *Proceedings of the thirteenth SIGCSE technical symposium on Computer science education* (pp. 92–96). ACM.

Kurland, D. M., & Pea, R. D. (1985). Children's mental models of recursive Logo programs. *Journal of Educational Computing Research*, *1*(2), 235–243.

Larkin, J. H. (1983). The role of problem representation in physics. In D. Gentner & A. L. Stevens (Eds.), *Mental models* (pp. 75–98). Hillsdale, NJ: Lawrence Erlbaum Associates.

Lee, V. R., Russ, R. S., & Sherin, B. L. (2008). A functional taxonomy of discourse moves for conversation management during cognitive clinical interviews about scientific phenomena. *Proceeding of the 30th annual meeting of the Cognitive Science Society* (pp. 1723–1728). Washington D.C.

Leonard, M. (1991). Learning the structure of recursive programs in Boxer. *Journal of Mathematical Behavior*, *10*(1), 17–53.

Levenick, J. R. (1990). Teaching recursion before iteration. *The Computing Teacher*, *17*, 12–15.

Levy, D. (2001). Insights and conflicts in discussing recursion: A case study. *Computer Science Education*, *11*(4), 305–322.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, *21*(1), 57–80.

Mandelbrot, B. B. (1983). *The fractal geometry of nature*. New York: W. H. Freeman.

McCloskey, M. (1983). Naive theories of motion. In D. Gentner & A. L. Stevens (Eds.), *Mental models* (pp. 299–323). Hillsdale, NJ: Lawrence Erlbaum Associates.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., Laxer, C., et al. (2001). *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students*. *ACM SIGCSE Bulletin* (Vol. 33, pp. 125–180).

McKinney, D., & Denton, L. F. (2004). Houston, we have a problem: There's a leak in the CS1 affective oxygen tank. *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 236–239). New York: ACM.

Mirolo, C. (2010). Learning (through) recursion: a multidimensional analysis of the competences achieved by CS1 students. *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (pp. 160–164). New York: ACM.

Murnane, J. S. (1991). Models of recursion. *Computer Science Education*, *16*(2), 197–201.

Newburger, E. C. (2009). *Home computers and Internet use in the United States: August 2000*. Washington, D.C.

Newell, A., & Simon, H. A. (1972). *Human problem solving*. New Jersey, USA: Prentice-Hall Inc.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental models* (pp. 7–14). Hillsdale, NJ: Lawrence Erlbaum Associates.

Oleson, K. E., Sims, V. K., Chin, M. G., Lum, H. C., & Sinatra, A. (2010). Developmental human factors: Children's mental models of computers. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* (pp. 1450–1453).

Parnafes, O. (2005). *The development of conceptual understanding through the use of computational representations*. University of California, Berkeley. *ProQuest Dissertations and Theses.*

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36.

Pea, R. D., Soloway, E. M., & Spohrer, J. C. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, *9*(1), 5–30.

Pirolli, P. L. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction*, *2*(4), 319–355.

Pirolli, P. L. (1991). Effects of examples and their explanations in a lesson on recursion: A production system analysis. *Cognition and Instruction*, *8*(3), 207–259.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, *39*(2), 240–272.

Resnick, L. B. (1983). Mathematics and science learning: A new conception. *Science*, *220*(4596), 477–478.

Russ, R. S., Sherin, B. L., & Drive, C. (2008). Reframing research on intuitive science knowledge. *ICLS'08 Proceedings of the 8th international conference on International conference for the learning sciences* (pp. 279–286).

Sanders, I., Galpin, V., & Götschi, T. (2006). Mental models of recursion revisited. *ACM SIGCSE Bulletin*, *38*(3), 138–142.

Scholtz, T., & Sanders, I. (2010). Mental models of recursion: Investigating students' understanding of recursion. *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (pp. 103–107). New York: ACM.

Schwill, A. (1994). Fundamental ideas of computer science. *Bulletin-European Association for Theoretical Computer Science*, *53*, 274–274.

Segal, J. (1995). Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, *22*(5), 385–411.

Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, *22*(1), 1–36.

Shackelford, R., McGettrick, A., Sloan, R., Topi, H., Davies, G., Kamali, R., Cross, J., et al. (2006). Computing curricula 2005: The overview report. *ACM SIGCSE Bulletin*, *38*(1), 456–457.

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

Siegler, R. S., & Crowley, K. (1991). The microgenetic method: A direct means for studying cognitive development. *American Psychologist*, *46*(6), 606–620.

Simon, H. A. (1956). Rational choice and the structure of the environment. *Psychological Review*, *63*(2), 129–138.

Singley, M. K., & Anderson, J. R. (1989). *The transfer of cognitive skill*. Cambridge, MA: Harvard University Press.

Smith, J. P., III, diSessa, A. A., & Roschelle, J. (1994). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences*, *3*(2), 115–163.

Soloway, E. M., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609.

Spohrer, J. C., & Soloway, E. M. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632.

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 97–101). New York: ACM.

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

The Joint Task Force on Computing Curricula. (2001). Computing curricula 2001. *Journal on Educational Resources in Computing*, *1*(3es).

Troy, M. E., & Early, G. (1992). Unraveling recursion, part II. *Computing Teacher*, *19*(7), 21–25.

Tucker, A. B. (1991). Computing curricula 1991. *Communications of the ACM*, *34*(6), 68–84.

Tulving, E., & Schacter, D. L. (1990). Priming and human memory systems. *Science*, *247*(4940), 301–306.

Turbak, F., Royden, C., Stephan, J., & Herbst, J. (1999). Teaching recursion before loops in CS1. *Journal of Computing in Small Colleges*, *14*(4), 86–101.

van Gog, T., Paas, F., van Merriënboer, J. J. G., & Witte, P. (2005). Uncovering the problem-solving process: cued retrospective reporting versus concurrent and retrospective reporting. *Journal of Experimental Psychology: Applied*, *11*(4), 237–44.

Vilner, T., Zur, E., & Gal-Ezer, J. (2008). Recursive thinking in CS1. *Proceedings of the ACM-IFIP IEEIII 2008 Informatics Education Europe III Conference* (pp. 189–197). Venice, Italy.

von Glasersfeld, E. (1995). *Radical constructivism: A way of knowing and learning*. London: Falmer Press.

von Wright, G. H. (1971). *Explanation and understanding*. Ithaca, NY: Cornell University Press.

Vosniadou, S. (2007). The cognitive-situative divide and the problem of conceptual change. *Educational Psychologist*, *42*(1), 55–66.

Wiedenbeck, S. (1988). Learning recursion as a concept and as a programming technique. *ACM SIGCSE Bulletin*, *20*(1), 275–278.

Wilcocks, D., & Sanders, I. (1994). Animating recursion as an aid to instruction. *Computers & Education*, *23*(3), 221–226.

Wu, C. (1993). *Conceptual models and individual cognitive learning styles in teaching recursion to novices*. University of Texas at Austin. *ProQuest Dissertations and Theses.*

Young, R. M. (1983). Surrogates and mappings: Two kinds of conceptual models for interactive devices. In D. Gentner & A. L. Stevens (Eds.), *Mental models* (pp. 35–52). Hillsdale, NJ: Lawrence Erlbaum Associates.

Youngs, E. A. (1974). Human errors in programming. *International Journal of Man-Machine Studies*, *6*(3), 361–376.

APPENDICES

Appendix 1

Götschi's (2003) categorization scheme for mental models of recursion

	Codes	Explanations
Active Flow	Copy	a new invocation with a new argument shown
	Loop	operation is done on the list element by element
	Not shown	only answer given or the trace is not detailed
	None	no recursion, one or two step evaluation
	Null	nothing can be concluded
	Algebraic	algebraic manipulation of function call
Base case	Stop	recursion stops once base case is reached
	Switch	once base case reached, switch from active to passive flow
	Check incorrect	incorrect test for base case
	Base omitted	operation at base case omitted
Passive flow	Copy	a partial solution is calculated at each level and returned to the previous invocation
	None	solution evaluated at base case
	Return values	each invocation's return value saved and used in calculating a solution
	Return problem	misconceptions about parameter passing and return value
	Operation changed	|| changed to + or × or combination, or order of operations Changed

Code Combinations (active flow, base case, passive flow)	Mental Models
Copy, Switch, Copy	Copies
Loop, Stop, None	Looping
Copy, Stop, None	Active
None, -, -	Step
Algebraic, -, -	Algebraic
-, -, return values/return problem/operation changed	Return value
"recognize syntactic segments as indications of recursive behavior and use their 'magic' ideas" (p.52)	Magic
"Show aspects of looping, algebraic and return value models, or were simply incomprehensible" (p.53)	Odd

Appendix 2

Adapted categorization scheme for mental models of recursion

Copies model

Keep invoking the method with new values until arriving at the base case, and then return to previous invocations until the original invocation.

Example (evaluation of the number-prefix method):

```
p (5, 4)
  5* p(5, 3)
    5* p(5, 2)
      5* p(5, 1)
        5* p(5, 0)
           1
        5*1=5
      5*5=25
    5*25=125
  5*125=625  → final answer
```

Active (combine all after base case) model

Keep invoking the method with new values until arriving at the base case, and then combine all the unfinished operations without showing the sequence of processing. Example 1 (evaluation of the number-prefix method):

```
p (5, 4)
  5* p(5, 3)
    5* p(5, 2)
      5* p(5, 1)
        5* p(5, 0)
  5* 5* 5* 5* 1 = 625 → final answer
```

Example 2 (evaluation of the list-prefix method):

```
q ([2, 1, 3, 8], 0)
  false && q([2, 1, 3, 8], 1)
         true && q([2, 1, 3, 8], 2)
               true && q([2, 1, 3, 8], 3)
                     true
  false && true && true && true = false  → final answer
```

Active (combine all along the way) model

Keep invoking the method with new values until arriving at the base case, and combine unfinished operations along the process of invoking.

Example (evaluation of the number-prefix method):

```
p (5, 4)
  5* p(5, 3)
    25* p(5, 2)
      125* p(5, 1)
        625* p(5, 0)
        625*1 = 625 → final answer
```

Active (combine last two) model

Keep invoking the method with new values until arriving at the base case, and then combine result of the base case and the result of the second last invocation.

Example 1 (evaluation of the number-prefix method):

```
p (5, 4)
  5* p(5, 3)
    5* p(5, 2)
      5* p(5, 1)
        5* p(5, 0)
            1
        5*1=5 → final answer
```

Example 2 (evaluation of the list-prefix method):

```
q ([2, 1, 3, 8], 0)
  false && q([2, 1, 3, 8], 1)
         true && q([2, 1, 3, 8], 2)
               true && q([2, 1, 3, 8], 3)
                     true
               true && true = true  → final answer
```

Active (revert to first) model

Keep invoking the method with new values until arriving at the base case, and then revert to the original invocation and finish evaluating statements after the recursive call.

Example (evaluation of the number-tail method):

```
g(10, 3)
   g(10, 2)
     g(10, 1)
        g(10, 0)
print "10*3=30"   → final answer
```

Active (revert to second last) model
Keep invoking the method with new values until arriving at the base case, and then revert to the second last invocation and finish evaluating statements after the recursive call.
Example (evaluation of the number-tail method):

```
g(10, 3)
   g(10, 2)
    g(10, 1)
       g(10, 0)
    print “10*1=10” → final answer
```

Active (output from all) model
Keep invoking the method with new values until arriving at the base case, and output at each invocation regardless how the recursive call and output statement are sequenced.
Example 1 (evaluation of the number-tail method):

```
g(10, 3)
print “10*3=30”
   g(10, 2)
   print “10*2=20”
        g(10, 1)
        print “10*1=10”
              g(10, 0)
Final answer: 10*3=30
         10*2=20
         10*1=10
```

Example 2 (evaluation of the list-prefix method):

```
q ([2, 1, 3, 8], 0)
  false && q([2, 1, 3, 8], 1)
        true && q([2, 1, 3, 8], 2)
              true && q([2, 1, 3, 8], 3)
                    true
  false  true  true  true  → final answer
```

Active (output from all but first) model

Keep invoking the method with new values until arriving at the base case, and output at each invocation except the original invocation.

Example (evaluation of the number-tail method):

```
g(10, 3)
  g(10, 2)
  print “10*2=20”
    g(10, 1)
    print “10*1=10”
      g(10, 0)
final answer: 10*2=20
              10*1=10
```

Active (base case result) model

Keep invoking the method with new values until arriving at the base case, and take the base case result as the final answer.

Example 1 (evaluation of the number-prefix method):

```
p (5, 4)
  5* p(5, 3)
    5* p(5, 2)
      5* p(5, 1)
        5* p(5, 0)
           1    →   final answer
```

Example 2 (evaluation of the number-tail method):

```
g(10, 3)
  g(10, 2)
    g(10, 1)
      g(10, 0)
        output nothing or print “10*0=0” → final answer
```

Example 3 (evaluation of the list-prefix method):

```
q ([2, 1, 3, 8], 0)
  false && q([2, 1, 3, 8], 1)
         true && q([2, 1, 3, 8], 2)
               true && q([2, 1, 3, 8], 3)
                     true    →    final answer
```

Bottom-up model

Evaluation starts from the base case up to the original invocation. A result is obtained at each invocation and used to evaluate the calling invocation.

Example (evaluation of the number-prefix method):

```
p(5, 0) = 1
p(5, 1) = 5*p(5, 0) = 5*1 = 5
p(5, 2) = 5*p(5, 1) = 5*5 = 25
p(5, 3) = 5*p(5, 2) = 5*25 = 125
p(5, 4) = 5*p(5, 3) = 5*125 = 625
```

Shortcut model

Final result is evaluated at the original invocation without invoking the method again with new values.

Example (evaluation of the list-prefix method):

```
q ([2, 1, 3, 8], 0)
false && q([2, 1, 3, 8], 1) = false → final answer
```

Step model

Only evaluate the method for once due to misunderstanding the recursive call as other operations.

Example (evaluation of the number-prefix method):

```
p (5, 4)
  5*p(5, 3) = (25, 15) → 40 → final answer
```

Function description model

Provide a description of the function of the recursive function, and the description does not fully reveal how the recursive function is executed.

Appendix 3

Mental models of recursion test

Name: ____________________________________ E-mail id: ________________________________

This quiz contains five questions, each on its own page. Each question has equal weight.

Please show your work for every question.

This quiz is closed-book, closed-note, closed-website, etc.

To facilitate using the results of this quiz in educational research, please do not put you name or E-mail id on any of the pages other than this cover page.

This quiz is pledged. You do not need to write out the pledge explicitly.

```
public static boolean f( int n ) {
 boolean result;

 if ( n == 0 ) {
  result = true;
 }
 else if ( n == 1 ) {
  result = false;
 }
 else {
  result = f( n - 2 );
 }

 return result;
}
```

If `f( n )` returns true, what do you know about `n`? `n` is ______________________

If `f( n )` returns false, what do you know about `n`? `n` is ______________________

```
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 );
 }
}
```

What output does invocation `g( 10, 3 )` produce? Please show how you arrived at your answer.

```
public static int p( int m, int n ) {
 int result;

 if ( n == 0 ) {
  result = 1;
 }
 else {
  result = m * p( m, n - 1 );
 }

 return result;
}
```

What value does invocation `p( 5, 4 )` return? Please show how you arrived at your answer.

```
public static boolean q( ArrayList<Integer> list, int i ) {
 boolean result;

 int n = list.size();

 if ( i == n - 1 ) {
  result = true;
 }
 else {
  int u = list.get( i );
  int v = list.get( i + 1 );

  boolean b = ( u <= v );
  result = b && q( list, i + 1 );
 }

 return result;
}
```

Suppose `a` is an `ArrayList<Integer>` whose element values are [2, 1, 3, 8]. What value does invocation `q( a, 0 )` return? Please show how you arrived at your answer.

```
public static void m( ArrayList<Integer> list, int i ) {
 int n = list.size();

 if ( i < n - 1 ) {
  int u = list.get( i );
  int v = list.get( i + 1 );
  boolean b = ( u == v );

  m( list, i + 1 );
  System.out.println( i + " " + b );
 }
}
```

Suppose `b` is an `ArrayList<Integer>` whose element values are [5, 7, 4, 8]. What output does invocation `m( b, 0 )` produce? Please show how you arrived at your answer.

Appendix 4

Frequencies of uncertain categorization for mental models of recursion

Untriangulated trace data

Mental Models (n=60)	Number-prefix	Number-tail	List-prefix	List-tail
Copies				
Active		7	10	6
Combine all after base case		1	7	
Combine all along the way				
Combine last two			1	
Revert to first		1		2
Revert to second last				1
Output from all		5		2
Output from all but first				
Base case result			2	1
Bottom Up		1		
Shortcut			1	
Function description				
Step		3	1	1
TOTAL	0	11	12	7
% Triangulation needed	0.0%	18.3%	20.0%	11.7%

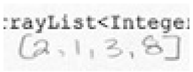
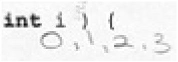
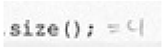
Triangulated trace data

Mental Models (n=60)	Number-prefix	Number-tail	List-prefix	List-tail
Copies				
Active		5	6	3
Combine all after base case			4	
Combine all along the way				
Combine last two				
Revert to first		1		1
Revert to second last				1
Output from all		4		1
Output from all but first				
Base case result			2	
Bottom Up		1		
Shortcut			1	
Function description				
Step		2	1	1
TOTAL		8	8	4
% Triangulation needed	0.0%	13.3%	13.3%	6.7%

Appendix 5

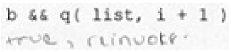
Participant No.7's interview transcript

Participant No.7's evaluation of the list-prefix method

Heeded Information	Transcript Segments	Associated Notes
q(a, 0)		
public static boolean q(ArrayList<Integer> list, **int** i) {	OK, so q is going to be my method. a, which is the list, my arraylist, which values are 2, 1, 3, and 8. And my int i is 0. So I wrote them on the top, just give myself a note, underneath the arraylist integer, 2, 1, 3, and 8, and wrote down that first, i is equal to 0.	 :rayList<Intege: [2,1,3,8]  int i) { 0,1,2,3
boolean result;		
int n = list.size ( );	Well since the list size is 4, I wrote beside int n, is equal to 4.	 .size(); = 4

if (i == n - 1){
 result = **true;**
 }
	So if n is equal to 4, then going to the next statement. The if statement, 0, which is i, is not equal to 4 minus 1, because that is 3. So I go to my else statement.	

else {		
int u = list.get (i);	And I get u, and u is therefore going to be first equal to 0 because it's list dot get i.	0
int v = list.get (i + 1);	And v is list dot get up once, so v is equal to 1.	1
boolean b = (u <= v);	And then you look at the Boolean value, as long as u is less than or equal to v, which 0 is less than or equal to 1, it is true.	true

result = b && q(list, i+1)
 }
	And I report the result, true, and also reinvoke the method of q list i plus 1. So i plus 1 now makes the original i, 1, and then you go back down,	 b && q(list, i + 1)
		 true, reinvoke·

return result;		
}		
q(a, 1)		
public static boolean q(ArrayList<Integer> list, **int** i) {		

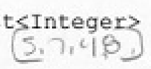
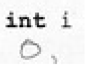
boolean result;
 int n = list.size ( );
	The list value is still 4, so n is still equal to 4.	

if (i == n - 1){
 result = **true;**
 }
	i which is now 1, is not equal to 3, and so my result is not true.	

else {	I go for the else statement.	
int u = list.get (i);	u is now 1.	1,

`    int v = list.get (i + 1);`	v is now 2.	2.
`    boolean b = (u <= v);`	And 1 is less than 2, so it's true.	true
`    result = b && q(list, i+1)`	And then I reinvoke the statement	true, reinvoke
`  }`	because the result is true, reinvokeing	
`  return result;`	the list i plus 1.	
`}`		
`q(a, 2)`		
`public static boolean`	So now i is equal to 2.	
`q(ArrayList<Integer> list, int i) {`		
` boolean result;`		
` int n = list.size ( );`	And i too is not equal to 4 minus 1,	
` if (i == n - 1){`	which is 3. So go down to else statement	
`  result = true;`	again.	
` }`		
` else {`		
`    int u = list.get (i);`	u is equal to 2.	2.
`    int v = list.get (i + 1);`	v is equal to 3.	3.
`    boolean b = (u <= v);`	My Boolean value is still true...	true
`    result = b && q(list, i+1)`	My result is true. And then once again	true, reinvoke
`  }`	reinvoke the statement, but it gives you	
`  return result;`	3 this time.	
`}`		
`q(a, 3)`		
`public static boolean`		
`q(ArrayList<Integer> list, int i) {`		
` boolean result;`		
` int n = list.size ( );`	And then n is still 4	
` if (i == n - 1){`	So i is actually this time equal to 3. So I	true
`  result = true;`	just write true.	
` }`		
` ……`		
` return result;`	And I return the result.	
`}`		

Participant No.7's evaluation of the list-tail method

Heeded Information	Transcript Segments	Associated Notes
`m(b, 0)`		
`public` **`static`** **`void`** `m(ArrayList<Integer> list,` **`int`** `i) {`	Ok, so, m (b, 0). b is that list 5, 7, 4, and 8. So I wrote that down underneath the arraylist integer list. And then i is supposed to be 0, so I put that beneath the i, just a reference to myself.	 t<Integer> (5,7,4,8)  int i 0,
`int` `n = list.size ( );`	And n is list size again, so it's equal to 4.	=4
`if` `(i < n - 1){`	And then i is equal to 0. The statement if i less than n minus 1, which in this case it is, 0 is less than 3.	3
`int` `u = list.get (i);`	The int value u is equal to list.get i, which is 0.	0,
`int` `v = list.get (i + 1);`	And v is equal to list.get i plus 1, which is 1.	1,
`boolean` `b = (u == v);`	And then Boolean value is asking whether or not the integer u is equal to the integer v, which is not, so I return the value false.	false
`m (list, i+1);`	**And then m list i plus 1, it would give you 1.**	1,

`system.out.println(i + "" +b)`
`}`
`}`
	And then it's asking you to print out i, space, and the Boolean value. So at first it will print out that for 0, if the value is 0, it would print out that, 0 is false, or, yea, 0, space, false. **And I don't know why I made it run through it again.** But the other two I wrote out, I wrote out the other two values down.	0 false.

`m(b, 1)`	1 false, because if I run through it again with, i plus 1, that would give you 1, which will also be false.	1 false.
`m(b, 2)`	And then here do it again, it would be 1 plus 1 which is 2, and that would also be false.	2 false.
`m(b, 3)`	And if run through it again, it would have 3, it wouldn't run through it.	

Appendix 6

Participant No.14's interview transcript

Participant No.14's evaluation of the number-tail method

Heeded Information	Transcript Segments	Associated Notes
g(10, 3)		
public static void int g(**int** m, **int** n) {	I started at g of 10 comma 3,	g(10,3) =
int product = m*n;	and so I got the product, m times n, so 10 times 3 is 30,	product = 30
if (n != 0){	and if n is not equal to 0, 3 is not equal to 0, so I did the if statement,	if n ≠ 0 ✓
g (m, n-1);	then you do g of 10, times, 10 comma 2, because n minus 1 is 2,	g(10, 2)

System.out.println (m+"*"+n+"="+product)
}
}
	(Skipped)	

g(10, 2)		
public static void int g(**int** m, **int** n) {	So I went through back at 10 times 2 at the top,	g(10, 2)
int product = m*n;	10 times 2 is 20,	product = 20
if (n != 0){	and then 2 is not equal to 0.	if n≠0
g (m, n-1);	So then I would do g of 10 1	g(10, 1) =

System.out.println (m+"*"+n+"="+product)
}
}
	(skipped)	

g(10, 1)		
public static void int g(**int** m, **int** n) {	So go back to the top.	g (10, 1)
int product = m*n;	10 times 1 is 10.	product = 10
if (n != 0){	1 is not equal to 0,	f n≠0 ✓
g (m, n-1);	so I had g of 10 comma 0,	g(10, 0)
System.out.println (m+"*"+n+"="+product)	(skipped)	

`  }`
`}`		

`g(10, 0)`		

`public static void int`
`g(`**`int`**` m, `**`int`**` n) {`
	and I went back to the top.	g(10,0)

` `**`int`**` product = m*n;`	10 times 0 is 0.	product=0 $\boxed{10 \times 0 = 0}$
` `**`if`**` (n != 0){`	And then, 0, this if statement does not apply anymore because n is equal to 0, and so I was like, I don’t know what to do now, because I thought that would be the end of the method.	

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

Participant No. 14's evaluation of the number-prefix method

Heeded Information	Transcript Segments	Associated Notes
`p(5, 4)`		
`public static int p(int m, int n) {`	Ok, so for this, I started at, I just plug in 5 for m, and 4 for n.	p(5,4)
`int result;`	-	-
`if (n == 0){`	So if n equals 0, which is not equal to 0, so I skip this statement.	-
`result = 1;`		
`}`		
`else {`	Then I got else. The result equals m*p(m, n-1), so that would be, so for p(5,4) would be 5*p(5,3).	p(5,4) = 5 · p(5,3)
`result = m * p(m, n-1);`		
`}`		
`}`	-	-
`p(5, 3)`		
`public static int p(int m, int n) {`	So then I was like, OK, I need to figure out what p(5, 3) is. So I went back through it (p method).	p(5,3)
`int result;`	-	-
`if (n == 0){`	Since 3 is not equal to 0	-
`result = 1;`		
`}`		
`else {`	The result is 5*p(5,2)	p(5,3) = 5 · p(5,2)
`result = m * p(m, n-1);`		
`}`		
`}`		
`p(5, 2)`		
`public static int p(int m, int n) {`	So I still don't know what p(5,2), so I gotta figure that out.	p(5,2)
`int result`	-	-
`if (n==0){`	Since 2 not equal to 0	-
`result=1;`		
`}`		
`else {`	I did p(5,2) equals 5*p(5,1)	p(5,2) = 5 · p(5,1)
`result=m*p(m, n-1);`		
`}`		
`}`	-	-
`p(5, 1)`		
`public static int p(int m, int n) {`	I still don't know what p(5,1) was, so I put, I was like, OK, I need to figure that out.	p(5,1)
`int result`	-	-
`if (n==0){`	-	-

`result=1;`		
`}`		
`else` `{`	So I did p(5,1) equals to 5 * p(5,0)	p(5,1) = 5 · p(5,0)
`result=m*p(m,`		
`n-1);`		
`}`		
`}`	-	-
`p(5, 0)`		
`public static int` `p(`**`int`**	And then p(5,0)	p(5,0)
`m,` **`int`** `n) {`		
`int` `result`	-	
`if` `(n==0){`	Since p(5,0) equals, since n equals 0, the result is 1	p(5,0) = 1
`result=1;`		
`}`		
`else` `{`	-	-
`result=m*p(m,`		
`n-1);`		
`}`		
`}`	-	-
p(5,0) = 1	So I finally figure out 1 was [the result of p(5,0)], so then I work backward because I knew that p (5,0) is 1	p(5,1) = 5 · p(5,0) p(5,0) = 1
p(5,1)	Then I knew that p(5,1) would equal 5 because 5 times 1	p(5,1) = 5 · p(5,0) = 5 · 1 = 5
p(5,2)	So then I knew that p(5,1) is 5, so I can figure out p(5,2) was 5 times 5	p(5,2) = 5 · p(5,1) = 5 · 5 = 25

p(5,3)
p(5,4)
	And then I did the same thing for all them until I got back to the top, where 5 times 125 because 5 times 25 was 125, so I got 625, that's the answer.	p(5,4) = 5 · p(5,3) = 5 · 125 = 625
		p(5,3) = 5 · p(5,2) = 5 · 25 = 125

Appendix 7

Participant No.56's interview transcript

Participant No.56's evaluation of the number-tail method

Heeded Information	Transcript Segments	Associated Notes
g(10, 3)		
public static void int g(**int** m, **int** n) {	Alright, so the first thing I did was assign the invocation of g 10, 3 to the parameters, so 10 would be equal to m and n would be equal to 3.	$g(\overset{m}{10},\overset{n}{3})$
int product = m*n;	And I actually think I forgot to calculate the product until after I did all these (statements below "int product=m*n"), but I will go through this first part before I go back to that.	
if (n != 0){	So the if statement says is n not equal to 0, and 3 is not equal to 0, so that's a yes, so then we go through with the if statement.	3 != 0 yes

g (m, n-1);
System.out.println
(m+“*”+n+“=”+product)
}
}
	And it says compute or invoke g of 10, which is still 10 and then n-1.	g(10,2)

g(10, 2)		
public static void int g(**int** m, **int** n) {	So then I go to g of 10 comma 2.	
int product = m*n;		
if (n != 0){	And then is n not equal to 0, so 2 is not equal to 0, yes.	2 != 0 yes
g (m, n-1);	So then it says in if statement invoke g of 10, n minus 1, which is g of 10, 1.	g(10,1)

System.out.println
(m+“*”+n+“=”+product)
}
}		

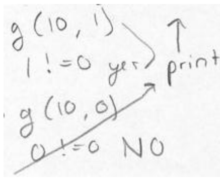
g(10, 1)		
public static void int g(**int** m, **int** n) {		
int product = m*n;		
if (n != 0){	So then 1 is not equal to 0, yes, it is not equal to 0.	1 != 0 yes
g (m, n-1);	So then the statement says, invoke g of m, n minus 1.	g(10,0)
System.out.println (m+“*”+n+“=”+product) }		

Participant No.56's evaluation of the number-prefix method

Heeded Information	Transcript Segments	Associated Notes
g(10, 0)		

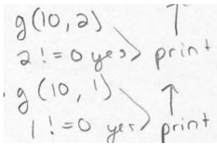
public static void int
g(**int** m, **int** n) {
int product = m*n;
	Which is now g of 10 0.	

if (n != 0){
g (m, n-1);
System.out.println
(m+“*”+n+“=”+product)
}
}
	And then is 0 not equal to 0, no, 0 is equal to 0.	0 != 0 NO

	So then it goes back to, uhu, the statement after it said this in the previous one.	
g(10, 1)		

public static void int
g(**int** m, **int** n) {
int product = m*n;
if (n != 0){
	So this one, or when g 10, 1-	

g (m, n-1);	-said to go to 10 comma 0, we did that, so that's complete. And now we go to next statement in the if statement.	

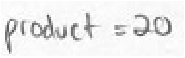
System.out.println
(m+“*”+n+“=”+product)
}
}
	And it says print out m times n equals product, so then we have printing, it prints out 10 times 1. And then this is when I went back calculate the product because I realized that I had read over that part, but it didn't make much difference when I was going through it. So then I did 10 times 1, so the product is equal to 10. So 10 times 1 is equal to 10 is what it would print. So then it goes back. I know, I finish doing g 10 of 1, so now I goes back to, when it run through g 10 of 2.	product = 10
		10 * 1 = 10
		 g(10, 2)
		2 != 0 yes
		print
		g (10, 1)
		1 != 0 yes
		print

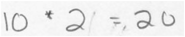
g(10, 2)		

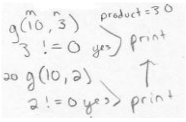
public static void int
g(**int** m, **int** n) {
int product = m*n;
if (n != 0){		

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

So this statement has completed, the g of m n minus 1 so now we can print m times n is equal to the product, so then it prints 10 times 2 is equal to 20, because that's what m times n equals to, and then so now I'm finished with g of 10 2. And now this goes back to where it was called upon in g of 10 3.


product =20


10 * 2 =20

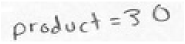


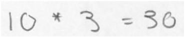
```
g(10, 3)
public static void int
g(int m, int n) {
 int product = m*n;
 if (n != 0){
  g (m, n-1);
```

So previously g of 10 3 had stopped after g of m comma n minus 1, which is g of 10 2,

```
  System.out.println
(m+"*"+n+"="+product)
  }
}
```

so now g of 10 3 can print out m times n is equal to the product, which would be 10 times 3 equals to 30, and then it's done


product =30


10 * 3 = 30