A Cognitive Model for Problem Solving in Computer Science

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Jennifer R. Parham
December 2009

Accepted by:
Dr. D. E. Stevenson, Committee Chair
Dr. Kenneth Weaver
Dr. Harold Grossman
Dr. Stephen T. Hedetniemi
Dr. Leo Gugerty

Abstract

According to industry representatives, computer science education needs to emphasize the processes involved in solving computing problems rather than their solutions. Most of the current assessment tools used by universities and computer science departments analyze student answers to problems rather than investigating the processes involved in solving them. Approaching assessment from this perspective would reveal potential errors leading to incorrect solutions.

This dissertation proposes a model describing how people solve computational problems by storing, retrieving, and manipulating information and knowledge. It describes how metacognition interacts with schemata representing conceptual and procedural knowledge, as well as with the external sources of information that might be needed to arrive at a solution. Metacognition includes higher-order, executive processes responsible for controlling and monitoring schemata, which in turn represent the algorithmic knowledge needed for organizing and adapting concepts to a specific domain. The model illustrates how metacognitive processes interact with the knowledge represented by schemata as well as the information from external sources.

This research investigates the differences in the way computer science novices use their metacognition and schemata to solve a computer programming problem. After J. Parham and L. Gugerty reached an 85% reliability for six metacognitive processes and six domain-specific schemata for writing a computer program, the resulting vocabulary provided the foundation for supporting the existence of and the interaction between metacognition, schemata, and external sources of information in computer programming. Overall, the participants in this research used their schemata 6% more than their metacognition and their metacognitive processes to control and monitor their schemata used to write a computer program. This research has potential implications in computer science education and software development through its understanding of the cognitive behavior used to solve computational problems.

Dedication

To my special Grammy, Marcia Adele D. Young, June 24th, 1932 - August 4th, 2006. For my son, Austin Keith Mocello, May 12th, 2007 -.

Acknowledgments

I want to begin by thanking Dr. Steve Stevenson for bringing me to Clemson University and believing in me every step of the way with or without tears:) His leadership, "out-of-the-box" thinking, determination to find answers, and passion for creating thinkers drive this research and taught me to always explore outside a vaccuum. This leads me to a special thanks to Dr. Lee Gugerty on my committee for spending a year of his time helping me carry out this research, and without his expertise, it would not have been possible. I want to thank Dr. Dean Bushey, Dr. Ken Weaver, and Dr. Steve Stevenson for listening to one semester on the relationship between schemata and cooking a turkey dinner. Without these discussions, I would have never discovered metacognition! In addition to the turkey dinner discussions, a big thank you to Dr. Bill Hanson, Dr. Bob Horton, Carol Wade, and all the other seminar people for picking my brain:) One of my biggest thanks goes to the wonderful Barbara Ramirez for helping me through the writing of this document and being a friend. She is one of a kind that every university needs. Thank you to the eleven participants in this research study who volunteered one hour of their time for $10, and thank you Rose Lowe and Dr. Robert Geist for letting me recruit these students from your class.

I want to thank all my friends and family for supporting me and listening to my discussions about metacognition and schemata, but a special thank you to my husband, Brian K. Mocello, for dealing with me daily through this research, keeping my head above water, and encouraging me to keep going. Without my husband's passion for my passions, this dissertation would not have been possible. I want to thank my parents, Laura and Stephen Sams, for instilling fortitude, courage, and independence. It is their hard-working ethics that have always inspired me. In addition, I am thankful for my grandfather's, John R. Young's, thought provoking questions about my research and for all my mother's sisters, Peggy Roberts, Robin Roberts and Lynne Parker, who influenced my love for teaching, compassion for others, and creativity. Also, I am thankful for all the free Lucky

Strike Jr. meals from my grandparents, Stephen and Judy Sams.

Lastly, I want to thank all the teachers and professors impacting my life in the mathematical and computing sciences. The teachers impacting my life prior to college include Philomeno Cone, Anita Cooley, and T. Sheriff. I want to thank Deloras Parks from Appalachian State University for introducing me to computer science and suggesting that I change my major from secondary education in mathematics to computer science. I am thankful for all my computer science professors at Appalachian State University for giving me a strong foundation in computer science that enabled me to succeed. As a master's student, I am thankful for all the research opportunities from Dr. Don Morton with the Arctic Region Supercomputing Center as well as his mentorship through my master's thesis, and I thank Dr. Mark Cracolice from the University of Montana's chemistry department for introducing me to higher order thinking and pushing me to pursue a Computer Science PhD. which led me to Clemson University.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Both computing professors and industry leaders acknowledge the need to give students in the computing sciences a real-world perspective through problems requiring a wide-range of skills [61]. However, even though the Association for Computing Machinery (ACM) and the IEEE Computing Society (IEEE-CS) recognize the necessity of teaching problem solving in the computing curricula [189], industry leaders believe that universities are not meeting this goal [56]. They believe that universities teach students to solve problems haphazardly by rummaging for a solution rather than teaching them to understand the processes required to reach a solution to a complex problem. Even though books used by universities to introduce computer science topics mention the importance of problem-solving skills in developing and analyzing algorithms, they do little to cultivate these skills [25] [104] [117] [87]. For example, these books may discuss problem solving in the first few sections, but subsequent sections emphasize writing computer code rather than the processes involved in developing an algorithm. Even though computer science professors might agree that the perfect student understands *Computers and Intractability* in its entirety [207], most universities do not use this book, which focuses on solving problems [73] as a foundation for the problem-solving knowledge required to be a computer science graduate, and in *Computing Curricula 2005*, the most recent curriculum published by the ACM, the authors acknowledge problem solving without providing instructors the details of how, what, and when to deliver this knowledge to students.

The reason for the disconnect between the problem-solving knowledge needed by industry and the skills taught at universities might be the lack of a unified view of what problem solving is, and this disconnect is further compounded by the fact that some researchers place importance on such artifacts as proofs and counterexamples resulting from solving a problem, rather than the history and process of how the solutions were developed [111]. Many computer science departments use assessment tools, such as standardized tests, pseudocode, and a computer program, to evaluate the ability to provide a correct answer to a problem, but these do not take into account the processes used by the student to arrive at a solution. This method of assessment and evaluation leads to frustration for both the professor and the student because of the inability to understand the possible errors leading to an incorrect result. The research in this dissertation focuses on discovering the processes involved in determining solutions to computing problems through the analysis of past problem-solving research, introducing a dynamic model to explain how people solve computing problems, and suggesting new testing strategies to capture this process in the computing sciences.

The theoretical framework for this research begins in 1910 with John Dewey's book, *How We Think*. Dewey posited that people construct thoughts through reflection and regulation [57], and in 1957, his work influenced George Polya, who validated the construction of thoughts in mathematical problem solving, recognizing the specific processes involved [166]. Dewey and Polya were among the first educational leaders to recognize problem solving as a set of processes rather than one process leading to a solution. Later, in 1985, Robert Sternberg used the idea of constructivism to develop his triarchic theory of intelligence, providing a detailed breakdown of the mental processes used to learn and solve general problems. Subsequently, he used this theory to compare expert and novice problem solvers [196]. Constructivism is now a well-established theory, in which individuals do not perceive the world directly but rather perceive interpretations of it, mediated by the interpretive frameworks developed [186]. The constructivist perspective serves as the foundation for two aspects of problem solving that are used as a focus in this research. The first aspect is the knowledge base and the second is metacognition, which controls and monitors thinking [65] [88]. According to Immanuel Kant, knowledge is represented through schemata, which form images and construct models using procedural rules [222]. Schemata are viewed as inert representations and structures that form the knowledge base, and the higher-order, metacognitive processes are responsible for choosing knowledge representations [197]. This research develops, evaluates, and assesses a new model representing the dynamic relationship between metacognition and knowledge representations to explain how people solve computer science problems.

More specifically, investigating the interactions between metacognition and schemata addresses the question of how people store, retrieve, and manipulate problem-solving knowledge and algorithms. In the model developed by this research, schemata represent the knowledge used to develop, analyze, and learn algorithms in computer science, and metacognition controls and monitors operations resulting in the storage, retrieval, and manipulation of dynamic schemata used for computer problem solving. Schemata include concepts, procedures, and patterns, which might be a design pattern for software development in computer science [72], and an example of a metacognitive process used to control schemata in computer science might select which design pattern to apply to the specific software being developed. To explore this issue of constructivism and the dynamics of problem solving, the research presented here utilizes the verbal protocol method in Ericsson and Simon's book, *Protocol Analysis* [62], to explain the relationships between students cognitive processes in solving a computer programming problem. After reaching an inter-rater reliability greater than eighty percent, this research established a common vocabulary of possible schemata, of results from operations on schemata, and of executive processes of metacognition used to solve problems in computer programming. The resulting vocabulary from this initial study was used to code the verbal protocols from second-semester and third-semester computer science students solving a first-semester computer programming problem, in order to determine the existence and uses of metacognition and schemata described by the model. Qualitative and quantitative analysis of the data collected from this methodology reveals that participants in this study used their metacognition 32 to 110 times, their schemata 60 to 133 times, and 0 to 25 external sources to determine a solution. To solve this problem, participants used their schema for writing code the most and the metacognitive process for inspecting their solution second most. The data from this study suggests that most students do not use the design schema for writing a program and metacognitive processes are responsible for determining when to write code as well as when to compile and execute the program. While some metacognitive processes and schemata are used more than others, on average, metacognition interacts with every schemata identified in the taxonomy. The results from this research may change the way people view the storage, retrieval, and manipulation of knowledge in computer problem solving, thereby impacting pedagogy, learning, and testing strategies at the university level.

Chapter 2

Background

Section 2.1 presents relevant research on problem solving to provide the basis for an investigation into the knowledge and metacognition needed for computer science. Sections 2.2 and 2.3 outline the primary studies in knowledge and metacognition, respectively. Research into memory as a container, reasoning for the justification, and education for development of knowledge and metacognition is presented in Sections 2.4, 2.5, and 2.6.

2.1 Past Research in Problem Solving

2.1.1 Historical Development

Until the 1890's, problem solving was defined in terms of a solution. In 1910, John Dewey defined a problem as anything that perplexes or challenges the mind to make the belief uncertain [57]. According to Dewey, belief is based on five steps of reflective activity.

1. The occurrence of a difficulty (sense problem)
2. Definition of the difficulty (specify problem)
3. The occurrence of a suggested explanation (propose solution)
4. The rational elaboration of an idea (evaluate solution)
5. Formation of a concluding belief (test solution)

Dewey described thinking as the process of reflection and regulation leading to a belief, and thought is a result of belief [57]. He explains this reflective activity as a transition between meanings and facts and connecting facts for thought. Induction is movement from particulars to a universal, connected view, and deduction is the process of making connections between universals and particulars. Dewey asserted that education advocates induction rather than deduction, i.e. the reasoning process.

Dewey applied and researched "how people think" in educational settings, and he presents thought as an artifact of thinking. Dewey explains that "I think so" implies that one does not yet "know so," i.e. knowledge requires thought (or belief), but thought is not knowledge and thinking is not knowing. In his investigation of thinking applied to problems, he discovered that thinking is a constructive process.

Expanding on Dewey's constructivist ideas of actively "building" knowledge and skills for problem solving, Bruner viewed problem solving as a systematic process involving readiness, a spiral organization, and going beyond the information given [30]. Bruner provides the following principles of constructivist learning

- instruction must be concerned with the experiences and contexts that make the student willing and able to learn (readiness)
- instruction must be structured so that it can be easily grasped by the student (spiral organization)
- instruction should be designed to facilitate extrapolation (going beyond the information given).

Cognitive structures, such as schemata, provide meaning and organization to experiences and allow the individual to "go beyond the information given" [31].

Parallel to Bruner's insights on constructivist learning, George Polya investigated how people solve mathematical problems. The mathematics education community is most familiar with Polya's work through his (1945/1957) introductory volume *How To Solve It*, in which he introduced the term "modern heuristic" to describe the art of problem solving [166]. Building on the work of John Dewey, Polya introduced four specific steps solving mathematical problems.

1. Understand the problem.
2. Find a connection between the data and unknown.

3. Devise a plan and take action on the solution.

4. Examine the results obtained.

Polya suggests that the result of a mathematician's work is demonstrative reasoning, a proof, but the proof is discovered by plausible reasoning, by guessing....

> Mathematical facts are first guessed and then proved, and almost every passage in this book endeavors to show that such is the normal procedure. If the learning of mathematics has anything to do with the discovery of mathematics, the student must be given some opportunity to do problems in which he first guesses and then proves some mathematical fact on an appropriate level [167].

Robert Sternberg adapts Polya's view of mathematical problem solving to general problem solving using his triarchic theory of intelligence [196], [197], [198]. In the late 1960s [88] and 1970s [64], metacognition became a topic of interest in psychology, and Sternberg incorporated metacognition into the constructivist view of problem solving. He stated that metacognition is needed in problem solving to select one or more representation/organization for knowledge and understanding [196]. His triarchic theory is built around three components: metacomponents, performance components, and knowledge-acquisition components. Metacomponents are the higher-order, executive processes used to plan, monitor, and evaluate problem solving. Performance components are lower-order processes executing commands issued by metacomponents, and knowledge-acquisition components are used for learning to solve problems.

In *The Triarchic Mind*, Sternberg outlines seven executive processes that makeup the metacomponents critical to problem solving intelligence [198].

1. Recognizing the existence of a problem.

2. Defining the nature of the problem.

3. Generating a set of steps needed to solve the problem.

4. Strategy selection: ordering the steps for problem solution.

5. Deciding how to represent information about the problem.

6. Allocating mental and physical resources to solving the problem.

7. Monitoring a solution to the problem.

About the same time, artificial intelligence (AI) began to have an impact on problem solving research, and many researchers used AI in conjunction with cognitive science to model problem solving methodologies. In 1972, Minsky introduced the concept of a frame, which is a data-structure for representing a stereotyped situation, such as being in a living room or going to a child's birthday party [138]. Parallel to Minsky's research, Newell and Simon developed the General Problem Solver (GPS) theory of human problem solving in the form of a simulation program. They used production systems, which are series of rules explaining the storage, retrieval, and manipulation of knowledge in problem solving [146]. Soon after, Anderson developed the ACT* theory of how people organize and use their knowledge, and he validated his theory using a programming language, LISP. His theory, also based on productions, has been used to explain how people program and learn new languages [8]. In 1978, Rumelhart began studying schemata and the processes, or modes of learning, used by them. Rumelhart identifies three modes of learning; accretion, structuring and tuning [180],and Schank and Abelson use script theory as the basis for a dynamic model of memory [184].

In 1992, Alan Schoenfeld wrote an extensive review of past problem solving research introducing both cognitive psychology and AI, entitled in "Learning to Think Mathematically" [186]. He pointed out five aspects of cognition based on research on how to solve problems:

1. a knowledge base
2. a problem solving strategy
3. control and monitoring
4. beliefs and affects
5. practices

A knowledge base includes chunks of information referred to as scripts, frames, and schema, and control and monitoring include self-regulation from the psychology, AI, and education literature. Schoenfeld provides a time-line graph displaying the differences in the time spent on each aspect, between an expert and novice solving a mathematical problem [186]. Schoenfeld demonstrated that the novice is unaware of or fails to use the executive skills used by experts, and he further addressed the need to teach these metacognitive skills for successful problem solving.

2.1.2 Expert vs. Novice Problem Solvers

More specifically, Robert Sternberg explored the differences between expert and novice problem solvers by explaining how the former classify a problem as a specific example of a particular class within a specific domain [199]. As a result, expert knowledge is connected and organized around important concepts, i.e. it is “conditionalized” to specify the context in which it is applicable, thereby supporting understanding and transfer to other contexts rather than only the ability to remember [50]. Consequently, experts are likely to understand a problem in terms of its structure and dynamics. As such, they are more likely to recognize what problems are analogous based on Polya’s research [173]. Once this context is established, experts group terms into chunks of information for easier access into their knowledge network [130].

According to Reisberg, chunking is an unscientific term for organizing elements of information into a non-fixed quantity grouped by relations; experts organize their knowledge around these patterns of information [173]. This concept is based on Miller’s ideas from 1956 that the working memory holds 7 plus or minus 2 chunks. In the late 1970’s, Hinsley explained how mathematics experts quickly recognize patterns of information, such as specific problem types that involve specific classes of mathematical solutions [50]. Hinsley, Hayes, and Simon found that even with a moderate level of expertise, subjects categorize problems in terms of their structure and type of solution rather than in terms of their superficial features [173]. These researchers believe chunking is achieved through schemata [186]. Similarly, Chi, Feltovich, and Glaser [36] found that experts in physics chunk various elements of a configuration that are related by an underlying function or strategy.

In the area of AI, an expert system is artificial intelligence software that uses data as input and incorporates concepts derived from expert knowledge in a field to provide problem analysis [120]. Dendral, Caduceus, and Mycin are examples of expert systems applied to the medical field. An expert system uses knowledge engineering to reason the same way as experts but independent of Sternberg’s research on expert problem solvers.

2.2 Knowledge

Content knowledge, or expertise, determines how information is encoded and organized [199], meaning that this prior knowledge influences how learners interpret and decide what aspects of this new information are relevant. Various types of knowledge such as facts, concepts, procedures, and experiences are required for understanding problems [92]. Problem solving and the resultant, learning, creates, or builds, new knowledge [148]. This knowledge is a continuum of knowing-how, or implicit knowledge, and knowing-that, or explicit knowledge [181].

Implicit knowledge is known as tacit, procedural, and operational knowledge. Sternberg uses schematic knowledge and tacit, or implicit, knowledge interchangeably [203]. This knowledge is organized as a recursive web of connected information and implies the ability to follow and create fundamental principles and systematic instructions that proceed from a problem statement to a solution [92].

In addition to implicit knowledge, explicit knowledge is needed for concepts, declarations, definitions, facts, and propositions. Explicit knowledge is known as conceptual, declarative, or propositional and includes relationships organized in a web-like network [92]. Cognitive psychologists believe concepts and definitions are stored as propositions resulting in true or false. John R. Anderson expands this idea using networks based on propositions having syntactical representation through relationships dependent on an object and agent, e.g. "Dogs eat bones" versus "Bones eat dogs" [7]. A person must resolve the propositional statement and relationships to use a concept from explicit knowledge.

According to Greeno, algorithmic knowledge is a continuum between explicit and implicit knowledge that links concepts and procedures using schemata, and competence for a principle concept may use many schemata to determine a solution [80]. The psychology and mathematics disciplines use schema theory as an explanation for procedural knowledge and its interactions with explicit knowledge during problem solving [92]. Past research exposes differences in the way individuals organize their knowledge leading to better problem solving skills [191].

2.2.1 Schematic Knowledge

Schemata are a valid explanation for algorithmic knowledge because the definition implies representation and structure through organization. Schema theory began in philosophy with Immanuel Kant. In Kantian philosophy, a schema represented knowing-how and solved the problem of how categories have "sense and significance" [221]. Kant defined a schema as the procedural rule by which a category or pure, non-empirical concept is associated with a mental image of an object [222]. Each category has one schema, and some schemata are shared by other categories in the same classification. The schema for a concept is the representation of a general procedure by which an image can be supplied for a concept.

The general application of schema theory is introduced by psychology and cognitive science, where a schema is a cognitive or mental structure implying a collective representation. People use schemata to organize knowledge and provide a framework for future understanding [49]. Schemata are used in logic to specify rules of inference, in mathematics to describe theories with infinitely many axioms, and in semantics to give adequacy conditions for definitions of truth. Examples of schemata include rubrics, stereotypes, social roles, scripts, worldviews, and archetypes. The architecture or structure of the mind was investigated through artificial intelligence using Schank's scripts, Minsky's frames, and Rumelhart's schemata.

Schemata have specific application in problem solving. In 1932, Sir Frederic Bartlett suggested that memory takes the form of schemata, which provide a mental framework for understanding and remembering information [17]. Bartlett investigated schemata used to solve problems through evaluating responses to open-ended questions [16]. For example, he gave students a beginning and ending number in a series and asked the students to fill in the series of numbers in the middle, i.e. 1...17. The various series of numbers supplied by the students revealed the different schema used to solve the problem and the mistakes commonly made in the solution.

In addition to Bartlett's experiments in the 1950's, Jean Piaget and Barbel Inhelder referred to problem-solving skills as schemata [98]. In Piaget's theory of development, children adopt a series of schemata to understand the world. His research included schemata development among children and the assimilation and accommodation of schemata. He used the example of an infant's push and pull schemata to demonstrate assimilation and accommodation. Assimilation is the process of integrating new information into an existing schema, and accommodation occurs when existing schemas or operations must be modified or new schemas are created to account for a new experience. Adaptation is the self-regulation process that operates through assimilation and accommodation [163].

Following Piaget and Bartlett's research, AI simulated the processes of schemata in problem solving using a computer to demonstrate and explain the psychology of how a human solves problems. In the 1970's, Rumelhart and Norman added two new processes to schemata called accretion and tuning [178]. In accretation, learners take the new input and assimilate it into their existing schema without making any changes to the overall schema. Tuning is when learners realize that their existing schema is inadequate for the new knowledge and modify their existing schema accordingly. Rumelhart redefined Piaget's accommodation using the term restructuring. Restructuring is the process of creating a new schema addressing the inconsistencies between the old schema and the newly acquired information [180]. John Anderson used Rumelhart's research on processes of schemata to develop the ACT* theory that uses neural networks in an expert production system to solve problems [8] [7].

Based on Anderson's theory on schemata, Sandra Marshall attached four components to schemata needed for mathematical problem solving [125]. She proposed a theory based on every schema having four main components that must all be active to have a fully functional schema [127] [124]. These four independent components and a description are as follows:

1. Feature recognition contains knowledge about the schema and specific situations for applying the schema.
2. Constraints are the set of rules and conditions for instantiating the schema.
3. Planning has the mechanisms for setting goals and subgoals used for implementation of the schema.
4. Implementation contains the actions and procedures for executing the goals and subgoals in the plan.

If any component is missing, a schema cannot be used to solve the problem, and a person would need more information to either trigger one of the properties or understand that the schema being used is not able to solve the given problem. Marshall developed the Story Problem Solver, SPS, assessment tool to show the existence of schemata in arithmetic problem solving [128].

2.2.2 Construction of Knowledge

Problem solving actively builds knowledge and skills using schemata, and this process is known as constructivism [47]. Solving problems in the sciences is built upon the scientific method, which is a constructive activity and not absorptive, where the solution is automatic for a person. John Dewey is thought of as the father of constructivism, and Jerome Bruner applied Dewey's constructivist ideas to learning. Bruner provides the following principles of constructivist learning: instruction must be concerned with the experiences and contexts that make the student willing and able to learn (readiness), instruction must be structured so that it can be easily grasped by the student (spiral organization), and instruction should be designed to facilitate extrapolation (going beyond the information given) [31].

Radical constructivism, derived from Piaget in the 1950s, is based on the assumption that children construct mental structures by observing the effects of their own actions on the environment [47]. Knowledge is constructed in the mind of the learner [24]. This knowledge is constructed as the learner organizes experiences around pre-existing mental structures or schemata. A constructivist model allows for a bi-directional flow of knowledge between students and teachers, and constructivism builds from the student's understanding towards the teacher's [24].

Students construct their knowledge by creating their own internal representations [93]. They bring information from their prior knowledge and modify the information they remember. There are pros and cons to constructivism because every learner constructs knowledge differently, but there are patterns to the construction of knowledge. Knowledge is created at three points: socialization, externalization, and internalization through metacognition [149] [148].

2.3 Metacognition

Metacognition has a directing or monitoring connotation depending on the context in which it is used. According to Flavell, metacognition is the control one has over their own cognition and learning [64]. Flavell made the distinction between metacognitive knowledge and metacognitive awareness. Metacognitive knowledge refers to explicit knowledge about our own cognitive strengths and weaknesses. Metacognitive awareness refers to the feelings and experiences one has when engaged in cognitive processes, such as retrieval. People must be confronted with contradictions or complexities that they must resolve by their own thinking and data gathering [65].

> Self-regulation is a cyclic process beginning with observations assimilated by current mental structures that drive behavior that has previously been linked to specific consequences (i.e.specific outcomes), and the outcome is determined to be either a good match or a poor match with accommodation needed [64].

J. T. Hart defines metacognition as the processes that allow an individual to observe, reflect on, or experience his or her own cognitive processes [88]. Monitoring informs the person of the state of their cognition relative to their current goal. Metacognitive monitoring includes feeling-of-knowing judgments, judgments of learning, ease-of-learning judgments, warmth judgments, and judgments of comprehension. Following Hart's research, Baker observed metacognitive processes including planning, monitoring, and modifying cognition at various stages in the acquisition of knowledge and skills [15].

Robert Sternberg defines metacognition as the higher-order, executive processes used to direct and monitor cognition. His triarchic theory of intelligence includes metacomponents (decide what to do, monitor while being done, and evaluate what is done), performance components (execute task), and knowledge acquisition components (learn how to perform task) [199]. Sternberg developed seven detailed roles of metacomponents (refer to Section.2.1.1). Metacomponents are executive processes activating performance and knowledge-acquisition components. Allen and Armour-Thomas constructed a test to validate six of Sternberg's metacomponents, and their results demonstrate that metacognition is an autonomous, multi-dimensional, general construct, and the six roles of metacomponents work through interaction [4].

2.4 Memory

According to Schoenfeld's diagram taken from Silver, the higher-order processes of metacognition are a part of working memory, and working memory uses stimuli, such as vision, hearing, taste, smell, and touch, from the sensory buffer and knowledge from long-term memory as inputs [186]. The metacognitive processes and mental representations in working memory use the input from the sensory buffers and long-term memory, as well as internal feedback from working memory, to produce a solution to the problem and store knowledge in long-term memory.

Memory is explained as a container, where metacognition and knowledge reside, divided into short-term and long-term memory [186]. Short-term memory is known as working memory, and is responsible for output from and storing knowledge in long-term memory [186]. Schoenfeld describes long-term memory as a permanent repository for knowledge, but the behavior and organization of knowledge for loading into short-term memory is still being researched. However, many researchers agree that knowledge in long-term memory is stored as chunks and organized as a neural network, or web of connected schemata [92].

2.5 Reasoning

The knowledge contained in memory is only knowledge if it is “justified, true, and believed” [205]. According to Plato, S knows p if and only if

1. p is true;

2. S believes that p;

3. S is justified in believing that p.

Reasoning is defined as drawing conclusions or inferences through the use of logical thinking through justification or support [46]. Therefore, reasoning is needed for the justification of knowledge.

In psychology, mental models are used for reasoning. According to Craik, a mental model is a dynamic representation of a corresponding system in the external world that simulates the continuous change of the external system [53]. Following Craik’s research, Johnson-Laird views mental models as the mechanisms which humans use to solve deductive reasoning problems [102]. He defines mental models as psychological representations of real, hypothetical or imaginary situations [103].

> “Thinking consists of the construction and use of models in the brain that are structurally isomorphic to the situations they represent [39].”

A schema is a type of mental model serving as a storage container for information used to represent problem solving knowledge.

2.6 Education

Problem solving and learning are two sides of the same coin in education. Learning is the process of acquiring knowledge, and problem solving is the process of using the acquired knowledge and results in learning. How one retrieves knowledge for problem solving reflects how one learns to solve problems [50]. Evaluating the way experts and novices solve problems demonstrates the differences in how people learn. Learning and problem solving retrieve, store, and manipulate knowledge.

Chapter 3

Details of Proposed Model

Based on the background presented in Chapter 2, this research proposes a cognitive model to illustrate the dynamics of solving problems in computer programming. Figure 3.1 provides a visual representation of the problem-solving paradigm indicating the transition from the problem statement to the cognitive model proposed in this research. As outlined in the previous chapters, problem solving begins with a problem, which is processed through one of the five senses, leading to an associated context that uses cognitive processing to establish it, to chunk the problem statement based on this context, to process these chunks, and to return to the problem statement. The cognitive processing proposed here defines the interaction between the metacognition and schemata needed when solving computational problems.

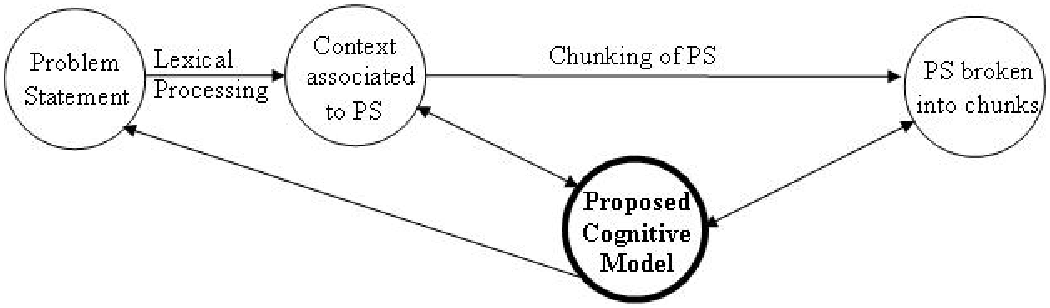


Figure 3.1: A diagram of the problem-solving process, beginning with the problem statement and its relationship to the proposed model.

While the literature serves as a foundation for this diagram, it expands past research to include the dynamics of schemata and their interactions with metacognition and external sources of information. Since this research is not concerned with the physiology of the brain, it does not consider its neurological make-up. Rather, it focuses on its apparent functionality when solving problems in computer science, as well as the relationships among the three components of metacognition, schemata, and external sources. According to Lev Vygotsky, people solving problems in a social setting will use their peers, books, and teachers, which are external sources of information, to help solve the problem [217] [218].

These three components, represented in Figure 3.2 by the three main boxes, act as initial and terminal states in the proposed cognitive model. Metacognition, the top box, serves as both an initial and a terminal state, controlling and monitoring the retrieval, storage, and manipulation of the schemata and the external sources of information. These processing units are represented by the blue and green boxes and the flow of data by the two sets of read/write arrows, while the schemata and external sources are only terminal states. In memory, knowledge or its representations are read and written by metacognition, and the information in an external source, eg. Google, is read and written by both schemata and metacognition. Schemata, as represented in the purple box, are templates for knowledge: External sources, shown by the orange box, can be many things such as books, peers, or the Internet. While some cognitive psychologists separate metamemory processes from metacognitive processes about individual beliefs [186], for the purpose of the model in Figure 3.2, metacognition includes both of these, as they function on the meta-level. In addition, the research reported here considers when knowledge is learned only in the broadest sense: that is, metacognitive and schematic knowledge either is in long-term or in short-term memory. The labels on the arrows between metacognition, schemata, and external sources are proposed operations initiating data flow, and at this time, this dissertation does not propose any other labels until collecting empirical evidence. The focus here is to capture the capability of a person to control and monitor data both from schemata and elsewhere. The executive, higher-order processes of metacognition are a collection of operations that can be categorized and processed as either controlling or monitoring, as represented by the green and blue box in Figure 3.2. These processes, as well as their operations, can occur simultaneously. For instance, a process controlling the recognition of a schema might issue a series of read operations, while at the same time, a process monitoring the results from a different schema issues a collection of other operations. Read and write operations transfer knowledge to and from metacognition, schemata, and external sources of information, represented by the two parallel data streams labeled read and write. However, if a schematic operation or series of operations fails,

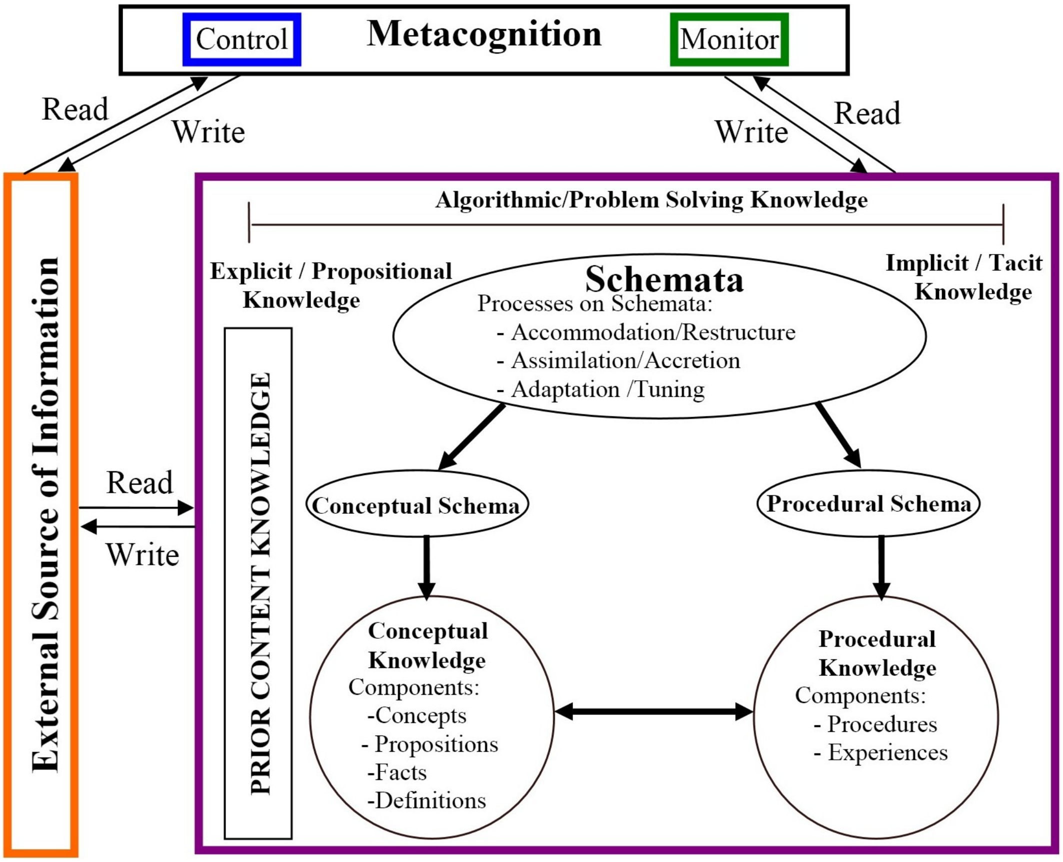


Figure 3.2: A cognitive model for solving problems illustrating the interaction among the controlling and monitoring operations of metacognition, the schemata representing prior content knowledge, and the information coming from external sources.

then metacognitive processes determine when, what, and how to access external sources to provide the missing information.

3.1 Flow Between Metacognition and Data

As shown in Figure 3.2, problem-solving knowledge is carried back and forth among schemata, metacognition, and external sources through operations and data connections. More specifically, a metacognitive operation of reading information from an external source can be sent at the same time as a scheduling operation for schemata or a metacognitive operation of reading schemata can occur simultaneously with an operation of writing a schema. Processing units control and monitor schemata and the use of an external source through a collection of operations specifying when and where to access the information needed. According to Schoenfeld, metacognition transfers knowledge between long-term and short-term memory [186], even though metacognition is stored in memory. As represented by Figure 3.2, the process of metacognition transferring knowledge from long-term to short-term memory can be explained through a series of events, such as those in a von Neumann computer where data/programs in memory are read and written by the central processing unit (CPU). However, in the proposed model, metacognition acts as the CPU reading and writing knowledge/schemata in memory. First, the control processing unit in metacognition issues a read operation containing the schema location in long-term memory, and then, the schema is transferred to metacognition, represented by the up arrow labeled "read". Next, the control processing unit issues a write operation to a short-term memory location, and the schema is transferred to short-term memory, depicted by the downward-pointing arrow labeled "write".

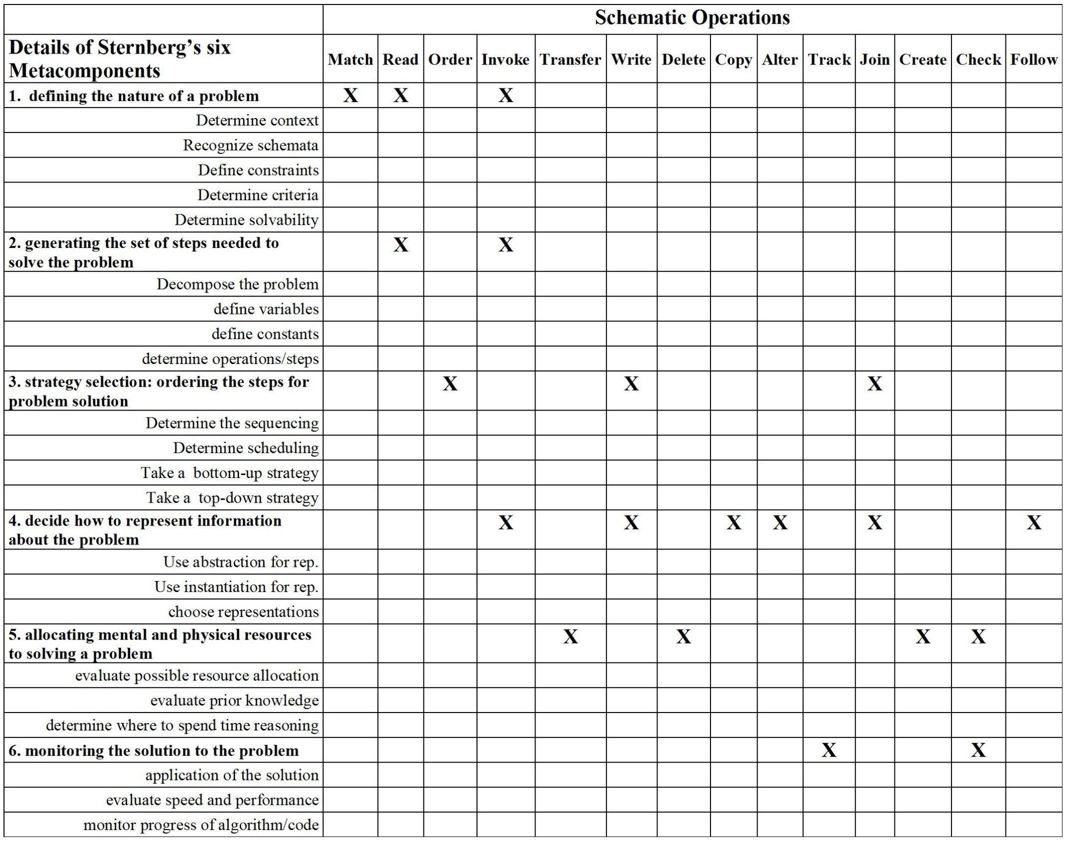
Going beyond the common, serial metacognitive processing used to transfer knowledge between long-term and short-term memory, more complicated examples include parallelism among processes and operations. For example, operations that control writing schemata to memory may occur at the same time as operations that monitor reading schemata from memory, and directly following these operations, other operations might be issued to control information being read from an external source. With this example, the initial operations from the control and monitor processing units are issued in parallel, but the operations used to control data from an external source are not; however, data may flow to and from different locations at the same time, based on the amount of data being written and/or read. Using Figure 3.2 as a visual aid, operations from the green and blue boxes are issued in parallel and schemata flow on both arrows between metacognition and the purple box. In addition, information may simultaneously flow on the up arrow between metacognition and the orange box, even though the operation controlling the flow was not issued at the same time as the others. Therefore, the order in which the transfer of data are completed cannot be determined, yet metacognitive processes handle the synchronization of schemata and other data through high-level operations.

3.2 Schemata Functionality

As discussed in Section 3.1, when solving problems, metacognition interacts with schemata, which are templates for algorithmic knowledge, through two processing units and shared operations. These operations are shared by metacognition and schemata to enable interactions and other dynamics for schema execution. Schemata use these operations to interact with other schemata and to determine the knowledge represented by the schemata. As seen in Figure 3.2, schemata are representations for conceptual and procedural knowledge, and the transformation from schemata to algorithmic knowledge is represented by the solid, black unidirectional arrows inside the purple box. Algorithmic knowledge, which is used to solve problems, lies on a continuum between explicit and implicit knowledge in memory, and schemata are templates and networks of templates representing the conceptual and procedural knowledge used in problem solving. The model described in Figure 3.2 is concerned only with the schemata representing prior content knowledge, which is the domain-specific knowledge from long-term memory used to solve the current problem, and similarly to metacognitive processes, the schematic processes, such as accommodation, assimilation, and adaptation discussed in Section 2.2.1, can use a collection of operations. For example, operations such as write, join, and check might be performed on schemata when assimilating new knowledge.

According to Sternberg, the executive processes of metacognition, known as metacomponents, are used during problem solving as well as while learning knowledge [198]. The model in this research uses six of Sternberg's problem-solving metacomponents and elaborates on these metacomponents with additional metacognitive processes unique to the computational sciences (refer to the left column in Table 3.1). For example, recognizing a schema is a detailed metacognitive process of Sternberg's metacomponent defining the nature of the problem. In Figure 3.2, the control and monitor processes of metacognition and the internal schematic processes share a common list of operations to use, those proposed in the top row of Table 3.1. Match, write, read, order, delete, create and copy are a few examples of schematic operations that might be used by metacognition as well as schemata. The storage, retrieval, and manipulation of schemata can occur simultaneously while using and learning knowledge, and they may require network connections and special operations for synchronization, scheduling, and error-handling. A schema may use other schemata in its internal network through operations such as communicate, schedule, invoke, and follow connections.

A schema is linked to other schemata through operations and connections. The model presented in this research follows Hiebert's observation that the more thoroughly information is understood, the stronger and more numerous are the connections [92]. The network of connections and the metacognitive and schematic processes determine whether a schema is used as a conceptual or procedural schema, respectively, leading to conceptual or procedural knowledge that can be trans-



	Schematic Operations													
Details of Sternberg's six Metacomponents	**Match**	**Read**	**Order**	**Invoke**	**Transfer**	**Write**	**Delete**	**Copy**	**Alter**	**Track**	**Join**	**Create**	**Check**	**Follow**
1. defining the nature of a problem	X	X		X										
Determine context														
Recognize schemata														
Define constraints														
Determine criteria														
Determine solvability														
2. generating the set of steps needed to solve the problem		X		X										
Decompose the problem														
define variables														
define constants														
determine operations/steps														
3. strategy selection: ordering the steps for problem solution			X			X					X			
Determine the sequencing														
Determine scheduling														
Take a bottom-up strategy														
Take a top-down strategy														
4. decide how to represent information about the problem				X		X		X	X		X			X
Use abstraction for rep.														
Use instantiation for rep.														
choose representations														
5. allocating mental and physical resources to solving a problem					X		X					X	X	
evaluate possible resource allocation														
evaluate prior knowledge														
determine where to spend time reasoning														
6. monitoring the solution to the problem										X			X	
application of the solution														
evaluate speed and performance														
monitor progress of algorithm/code														

Table 3.1: Examples of possible interactions between metacognitive processes and schematic operations

ferred among schemata, metacognition, and external sources. Conceptual knowledge includes facts, definitions, concepts, and propositions; whereas, procedures and experiences are classified as procedural knowledge. This research acknowledges episodic knowledge, but for the purposes of this model, experiences from episodic knowledge used in problem solving are classified as procedural knowledge. The model in Figure 3.2 uses schemata as representations and as structures of knowledge. A knowledge structure is a network of knowledge representations. Just as computer programs have data representations, data structures, and operations, a schema can contain knowledge representations and structures consisting of a single schema or networks of schemata, as well as schematic processes and operations. The representations in a schema determine how general or specific the schema is. Depending upon the problem, metacognition and schemata dynamically interact to determine the representations and operations responsible for the storage, retrieval, and manipulation of knowledge.

In addition to connections and operations that determine the representation and structure of schemata, there are common properties among all schemata, such as feature recognition, constraints, planning, and implementation [124]. The model presented in this research includes another property, which is termed criteria. Criteria denotes the set of principles used to determine whether a schema is instantiated. The recognition of a schema is a result of a metacognitive process reading the feature recognition property, and activation is a result of a successful match between the context of the problem and the criteria property of a schema. Ultimately, this property is used by metacognition as the primary reason for choosing a schema. For example, a person may read a sorting problem in computer science and immediately recognize the bubble sort, quick sort, and merge sort schemata, and then, after determining the context of the specific problem, the person may use this information and the criteria property of all recognized schemata to metacognitively choose, for example, the bubble sort schema. The constraints property and planning property of schemata are used to generate the steps needed for a solution and a strategy for ordering the steps; the execution of a schema is based on the implementation. Metacognition controls and monitors the recognition, choosing, ordering, and execution of a schema using these properties, knowledge representations, operations, and data connections. Schemata and metacognition may use external sources of information in order to complete the recognition, choosing, ordering, and execution of schemata, in case of errors or missing knowledge.

3.3 External Sources

A schema may use an external source of information to fill in data that is constant and does not need to be permanently stored. For example, a person might not store the gravitational constant in memory; rather, he/she may have an operation that reads this value from the nearest source of information. In addition, when a schematic operation or series of operations fails, metacognitive processes might use an external source of information to fix the error. Not all errors result in the use of an external source, such as those refered to as misconceptions. Misconception implies an error in conceptual knowledge; the model presented in Figure 3.2 captures more than just conceptual knowledge. Therefore, misconceptions, as well as other errors in metacognition and schemata, are included in the proposed model, and these errors can occur in the metacognitive control and monitor processes, in a schematic operation, in a schema property, in conceptual and procedural knowledge, and in the network. Generally, if these errors do not cause a disruption in a person's cognition, then the errors are not fixed intentionally. However, if a person metacognitively notices one of these errors, then he/she may decide to use an external source of information. Just as schemata are a part of memory and evolve over time by manipulating existing schemata and storing new schemata, metacognition is also a part of memory and can evolve through the manipulation of existing metacognitive processes and storing new metacognitive processes. This explains why a person may need an external source to solve a problem one time, and the next time the person does not use an external source of information to solve the same (or similar) problem.

Books, articles, the Internet, teachers, and peers are examples of external sources of information. These sources can provide the missing information needed to solve a problem. However, the information retrieved from an external source may not be true and may either be learned permanently or lost right after its use, depending on whether the information is integrated into working memory and transferred back to long-term memory. Metacognition may use an external source of information in any of the following cases:

- **Missing a Schema:** A schema for a specific problem cannot be selected because the schema is not stored in memory. This is the result of attempting to check, match, or execute a missing schema.

- **Missing a Connection:** A connection does not exist between schemata. This results in a failure to follow a connection from one schema to another.

- **Missing a Property:** A missing schema property cannot be accessed or used. Failure to read properties of a schema results in the inabilty to use the schema.

- **Mismatch in Results:** There is a mismatch between the expected and actual results of a schema. The tracked behavior of a schema does not match the expected behavior.

3.4 Hypothesis

Based on Figure 3.2 and the description of the proposed model presented in this chapter, the following hypothesis is proposed:

> “Figure 3.2 is a valid cognitive model for solving problems in computer programming, where metacognitive processes are responsible for controlling and monitoring the need for

> external sources of information and the schematic operations that result in the storage, retrieval, and manipulation of schemata, which are representations of knowledge marked by continuous change, activity, and progress."

The cognitive model in Figure 3.2 provides a visual representation of the roles of metacognition and schemata in computer problem solving. The goal of this research is to establish interactions between content schemata and metacognitive processes as an integral part of the discipline, using the model as a foundation for establishing the details of how people store, retrieve, and manipulate knowledge. This research has the potential of improving computer science learning and pedagogy through a better understanding of the operations on schemata as well as the validation of metacognitive processes, such as Sternberg's six metacomponents. Ultimately, the significance of this research is to validate a model and develop a vocabulary for educators to better understand, teach, and assess individual and groups of students' behavior while solving problems in computer science. This has the potential to lower frustration between the educator and the student and to positively impact learning through a focus on pedagogical strategies emphasizing the process of storing and retrieving information.

Chapter 4

Research Design and Methods

To address the issue of retrieving and storing information while solving computational problems, this study uses verbal protocols from first-year and second-year computer science students as they solve a problem requiring prior content knowledge about the array data structure, looping, and other basic concepts used to write a computer program. Verbal protocols include all methods for obtaining oral and/or written reports of thinking, such as interviews, thinking aloud while solving a problem, categorizing problems, reflective thinking about solutions or problem statements, and asking participants to describe or explain their verbalizations. Since different methods yield different results, several can be combined. For example, a mixed-method protocol requiring thinking aloud while solving a problem combined with interviews and reflective reasoning helps capture parallel thoughts.

This methodology, well-known in many disciplines including cognitive psychology, physics, chemistry, mathematics, education, business, artificial intelligence (AI), and human-computer interaction (HCI) [36], [90], [215], [141], [42], [20], [67], [146], [182] has yielded innovative assessment tools in chemistry [213] and physics [90]. For example, in physics, Hestenes *et al.* used verbal protocols to develop the Force Concept Inventory (FCI), an assessment tool measuring student misconceptions about motion and force [90]. The three-year study initiating the creation of this instrument [86] was based on John Clement's 1982 publication, *Students' Preconceptions in Introductory Mechanics* [41], and his research, in turn, was based on two articles by a physicist and an educator studying student understanding of velocity and acceleration through written answers and verbal protocols [214] [215]. To create the multiple-choice test preceding the FCI, McDermott, Clement, and Hestenes collected more than five years of data by videotaping/recording students writing answers and interviewing them to determine the reasoning behind these answers. This record of students solving problems captured their stepwise solutions for the alternative answers on the multiple-choice tests, with the interviews capturing their reasoning and misconceptions. More recently, John Clement and faculty from computer science held a research panel discussing misconceptions in discrete mathematics [6], which resulted in an NSF grant to collect verbal protocols to aid in the development of a new assessment tool based on the FCI [5].

As computer science researchers have begun to recognize a need for domain-specific transcripts and analysis [5] [37] [160], they have adapted Ericsson and Simon's guidelines for producing and analyzing verbalizations in *Protocol Analysis.* This book defines verbalizations resulting from thinking aloud as representations of subsets of sequences of thoughts reflecting the details of the retrieved information even though these thoughts do not describe why this knowledge was chosen [62]. The authors acknowledge other methods for measuring cognitive processes, such as EEG and eye tracking; however, none claim to capture all thoughts, including thinking aloud. Ericsson and Simon posit that people may have unconscious thoughts while they are silent, that thinking aloud captures only sequential thoughts, and that the combination of verbal protocols, such as interviews and experimenter interjection, with other methodologies like eye-tracking help capture unconscious and parallel thoughts. In addition, researchers tend to criticize verbal protocols for disrupting normal cognitive behavior, but any "online" method which observes a person while performing a task can be disruptive to normal behavior [59].

After comparing various methods, verbal protocols were determined to be an appropriate methodology for yielding details about metacognition and schemata used in computer science problem solving, and the research presented here is based on Ericsson and Simon's protocol for explaining the details of student cognitive processes when solving a complex computer science problem. To investigate their metacognition and schemata more accurately, this research collected verbalizations from participants using a combination of thinking aloud while solving a problem, providing participants with information when asked, and conducting off-line interviews after the participant reached a solution. The mixed-method approach employed here addresses the details of the dynamics of cognition required to explore the relationship between metacognition and schemata during problem solving in computer science. At this time, this research is concerned only with conscious thoughts as well as parallel thoughts captured by the interview; however, it recognizes unconscious thoughts and does not dismiss the possibility of their being a part of the explanation for the results during periods of silence.

4.1 Participants

The participants included students enrolled in either Computer Science II (CPSC 102) or Data Structures and Algorithms (CPSC 212) at Clemson University who volunteered one hour of their time in exchange for ten dollars from the experimenter (under IRB consent #IRB2008-170 entitled “Test of Computational Thinking”) to participate in this research study. After being given a brief introduction to the experiment, the participants were recruited by passing a calendar around the two classes asking students to sign up for a date and time when they were available to participate. This yielded a sample size of eleven, and if offered more money, for example twenty dollars, the sample size may have increased but so would the cost. On the other hand, if the experimenter was an instructor of a course, then the students could have been recruited through an extra credit or homework option in lieu of a traditional assignment. After volunteering to participate in this research study, participants received an email welcoming them to the study and providing them with the IRB informational letter (see AppendixC). Each participant was sent an email 24 hours prior to his/her sign-up date reminding them of their time and the location, Brackett Hall, Room 317A, along with a demographics sheet to minimize confounding variables such as years of professional programming experience versus university programming experience. This quiet, well-lighted room without any distractions is located in the Psychology Department and dedicated to conducting verbal protocols. The time of the day for the experiment varied based on each participant and the room availability; the physical, mental, and emotional state of the participant was ignored. However, future studies can capture this information on the demographics sheet to establish the differences in metacognition and schema use under different conditions due to external variables such as caffeine, sleep deprivation, empty stomach, and stress.

4.2 The Pilot Study

A pilot study was conducted to ensure analyzable data based on the initial demographic sheet and the problem selected. For example, the first participant was given the original demographics sheet (see Appendix A), but after completing the verbal protocol, the demographics sheet was updated to include the number of English speaking years for accuracy in transcription and analysis. In addition, the demographics sheet was modified to capture information on ethnicity, grade in past CPSC course at Clemson, and whether the participant is a first-generation university student (see Appendix B). Prior to recording, each participant in this study was read the IRB Informational Letter, which was sent in the initial welcoming email (see Appendix C), followed by a brief explanation of a verbal protocol using adding eleven plus eleven as an example; i.e., the experimenter's verbal protocol was "One plus one, two. One plus one, two. Twenty-two." After informing the participants that they could ask a question at any time, the researcher supplied every participant with a computer capable of browsing the web, accessing old programs, and writing new programs as well as a blank piece of paper, pencils, pens, textbooks on Java, C, and C++ languages, and a paper-copy of the problem statement.

In this study, in order to capture the details of metacognition and schemata in the transcript, piloting the selected problem was important. A problem too simple or too hard will not yield as much verbalization as a problem complex enough to stimulate thought [62]. For instance, if the research question involves the carry schema used in addition, then the problem needs to be difficult enough for the participant to verbalize details about it. The experimenter's verbal example of eleven plus eleven was too simple to contain any information regarding the carry schema. For the purposes of this research, the problem was chosen to provide details about the looping and array schemata as well as the interactions within and among these schemata, metacognition, and external sources of information.

Before the students were instructed to begin working on the problem, the experimenter read the following version of Chapter 8 Programming Project 1 from *Problem Solving and Program Design in C* [87]:

> Write a program to take two numerical lists of the same length ended by a sentinel value and store the lists in arrays x and y, each of which can hold a maximum number of 10 elements, and populate both lists by entering the actual data values from within the program. Let n represent the actual number of data values in each list. For example, x and y can hold up to 10 elements, but the two arrays may only contain 5 actual data values, i.e. n=5. Store the product of corresponding elements of x and y in a third array z, also of maximum size 10. Print the contents of the arrays x, y, and z. Then compute

and print the square root of the sum of the items in z.

This problem was given to the initial five students with various backgrounds, and the resulting data contained enough detail among these participants to continue using in the research study.

4.3 The Research Study

As in the pilot study, every participant in the research study was instructed to meet in Brackett Hall, Room 317A on the date and time specified on the volunteer sheet. They were read the IRB informational letter, given an example verbal protocol, and supplied with texts, paper, pencils, and a computer. The verbal protocols were recorded on a desktop computer in the room using Morae software and a microphone. Morae software records a person's voice and computer actions, resulting in a video of the participant writing computer code and talking aloud simultaneously while solving a computer science problem. Due to the absence of a video camera and eye tracking software, participants were instructed to verbalize their actions performed off the computer such as designing, reading the problem statement, and using textbooks. The student could use books, the internet, scratch paper, the interviewer, and old programs when he/she could not remember information or simply did not know the answer. The interviewer answered questions and provided hints only when the student specifically asked the interviewer for help or appeared to be struggling on the same thought for longer than 5 minutes, and participants were reminded to think aloud if silent for more than 10 seconds to increase the number of thoughts captured.

4.4 Transcription of the Data

The original format of the data was a Morea video clip accessible only by using the Morae software, which transferred the video clip into a Windows movie (.wmv) file and a comma-separated values (.csv) file containing the elapsed time, the computer page/program, the keystrokes, the application changes, and the internet searches (see Appendix F). Using the Morae .csv spreadsheet and the coding guidelines in Appendix H, columns A through E in the Level 1 spreadsheet were created to capture the details about participant computer actions and keystrokes (see Appendix H). The verbal statements were manually transcribed into text, while viewing and listening to the movie file, and entered into column F next to the corresponding computer actions in the Level 1 sheet (see Appendix G). The recorded data from the eleven participants consisted of 298 minutes of audio and video, but due to the inability to understand the first participant, this verbal protocol was not transcribed or analyzed. This yielded 241 minutes and 32 seconds of audio/video to transcribe and analyze as well as 357 pages of .csv data to transfer to columns A through E in Level 1. For this research study, the average initial setup time for columns A through E was 2 hours, which depended on the participant's total time to solve the problem and the number of computer actions during that time. Initially, transcribing the data averaged one hour for every 2 minutes of participation, but the experimenter reduced the transcription time by half after three participants. After four more participants, the transcription time was again reduced to one hour for every 6 minutes of participation, and the Level 1 transcript in columns A through E for the 10 participants resulted in 200 pages of data to analyze and code. Even though transcription and analysis might take less time using only computer actions, the verbal transcript is needed to capture the relationship between schemata and the metacognitive processes.

4.5 Analysis of a Transcript

After transcribing the 10 participants, a process of joint and individual coding of transcripts was used to establish the reliability for the vocabulary and transcript analysis. Initially, the researchers analyzed the transcripts for general commonalities using the experimenter's raw observations after each verbal protocol (see Appendix Q) to develop the initial coding document for metacognitive processes and schemata (see Appendix I). To code the high-level and low-level domain-specific schemata of a participant, the problem statement was divided into the goals and subgoals needed for the solution (see Appendix D). The starting time for the participant's implicit and explicit goals and subgoals were recorded in columns G and H of the Level 1 spreadsheet as well as the time when the participant finished coding and completing his/her answers. Errors in the goals and subgoals were recorded in column I, along with the participant's detection and correction of these errors. The final coding document in Appendix H is a result of seven versions of joint and individual coding/recoding to reach an acceptable inter-rater reliability greater than 80% on the metacognitive processes and programming schemata.

Jointly, the researchers created a list of goals and subgoals for the problem (see Appendix D), as well as an initial coding scheme based on the patterns among the transcripts, the metacognitive processes proposed in Table 3.1, common programming strategies, the domain-specific schemata need to solve the problem, and Bloom's taxonomy (see Appendix I). After applying this first coding document to the ninth participant's transcript, the researchers met for 1.5 hours discussing their differences in the first 8.5 minutes of participant 9. This discussion led to modifications, additional examples, and clearer definitions, all of which can be seen in Appendix J. The most significant modifications included renaming the terminology for the programming strategies and omitting Bloom's taxonomy due to its inability to capture or add details about metacognition and schemata in computer science problem solving. The ninth participant was recoded using version 2, followed by a 3-hour meeting discussing disagreements on the first 8.5 minutes of the recoding. A direct result of this meeting was the addition of a code, learn syntax by guessing and compiling, as well as details to the goals and subgoals (see Appendix K). The goals and subgoals were divided into columns G and H, and the start time for beginning code, finishing code, and evaluating code for goals and subgoals was added to the document. Using version 3 of the coding scheme, participant 9 was recoded again, and the researchers discussed their disagreements on the first 14 minutes of the transcript. Version 4 added examples and improved definitions (see Appendix L), and after recoding participant 9, the researchers reached a joint agreement on the analysis of the first 14 minutes of this student. Reaching this agreement on the vocabulary took approximately 150 hours over two months.

The next two months consisted of the researchers individually coding two students to reach an acceptable inter-rater reliability on the coding scheme for the metacognition and the schemata used in this computer science problem. The first half of the transcript from the fourth participant was chosen at random for independent coding. Using version 4 of the coding document, the researchers reached a 71% agreement on 60 codes from 1:53-8:33 minutes of this participant. This inter-rater reliability is lower than the accepted 80% or higher in psychology leading to another 2-hour meeting. The topic of this meeting was the development of a new code to capture the participants' diagnosing errors before fixing them and was to improve definitions (see Appendix M). The first half of the fourth participant was recoded jointly using version 5, and then, participant 6 was randomly chosen for the second individual coding. The first 13 minutes of this participant yielded 120 codes, and the raters agreed on 60% of them across columns G through K. This inter-rater reliability was lower than the initial reliability, perhaps because of the addition of codes without clear definitions and/or a longer transcript with twice as many codes as participant 4. Based on the disagreements and 10 more hours of meetings, subcodes were added to the diagnose code to capture the different diagnosing strategies such as recognition, elimination, and backtracking and to the planning code to include general scheduling of activities (see Appendix N) as well as a second analysis of the problem goals and subgoals (see Appendix E). This sixth version of the coding document was re-applied to the first half of participant 6, and the researchers individually coded the remaining 13 minutes of this participant. Of 96 codes, the raters agreed on 64, improving the inter-rater reliability of the coding document to 67%. Most of the disagreements in this third inter-rater reliability coding were due to subcodes rather than the high-level codes. Therefore, the coding document was simplified by combining and removing subcodes (see Appendix H), and the individual coding from the last 13 minutes of participant 6 were recoded using this version, resulting in an 85% reliability for the vocabulary used to code metacognitive and schematic processes. Using the final version of codes, columns G through K from the first half of participant 6 was recoded (see Appendix G), and coding the errors, metacognitive processes, domain-specific schemata, and the use of external sources of information averaged the experimenter one hour for every 5 minutes of participation in the remaining transcripts.

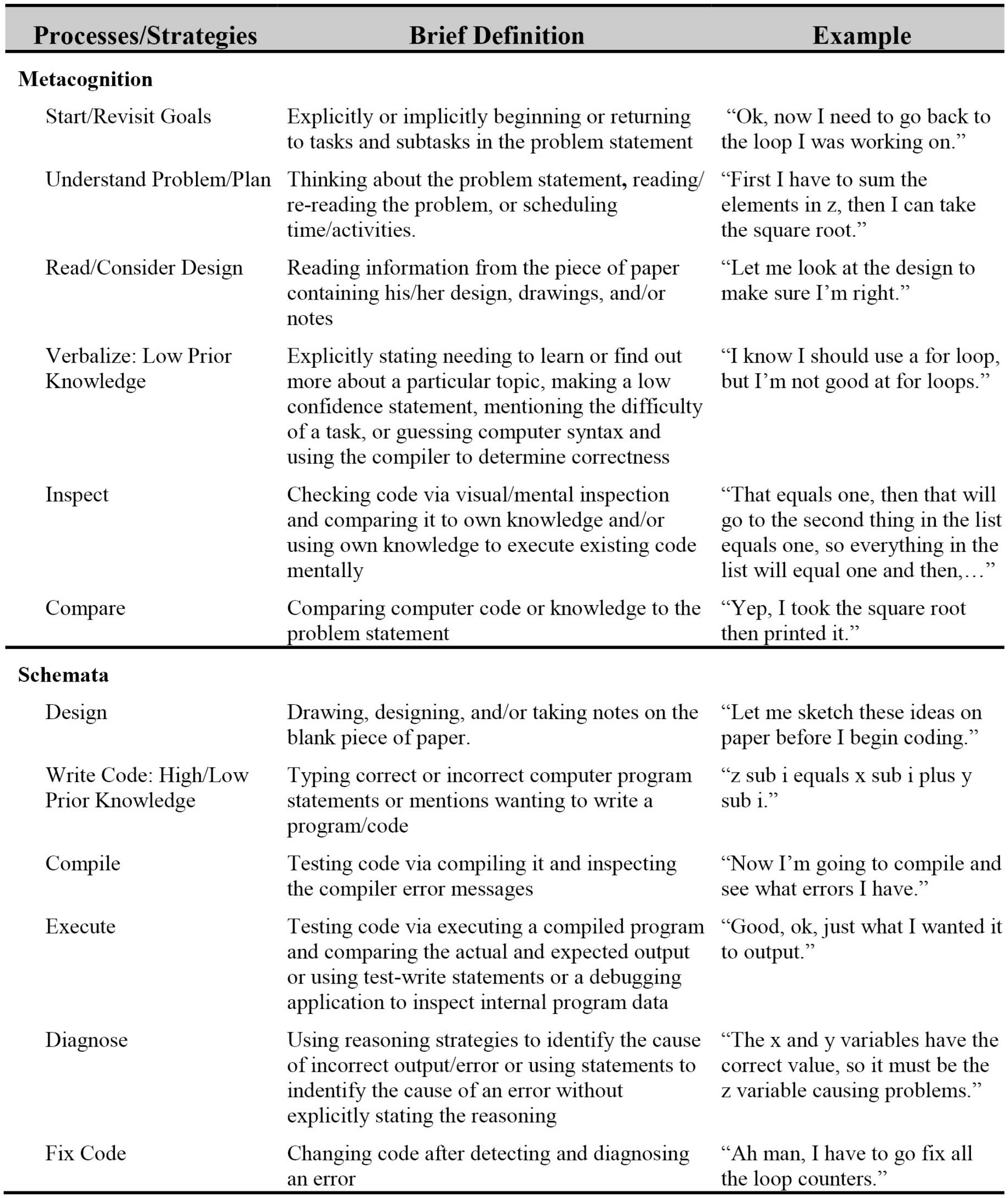
Chapter 5

Results and Discussion

After reaching an agreeable inter-rater reliability, a vocabulary for metacognitive and schematic processes was established to determine each participant's Level 1 coding (see Appendix G). This vocabulary led to the development of Level 2 graphs capturing participant behavior among goals and their movement through the model proposed in Figure 3.2. The patterns and interactions among these Level 2 goals, errors, metacognitive processes, and schematic processes were then analyzed for commonalities and differences, and these results compared to the participant's grade on the problem, to their demographic information, and to Figure 3.2. The analysis of the data required a taxonomy of mental processes, Level 2 graphs, and grading schemes for the problem.

5.1 A Taxonomy of Mental Processes

The vocabulary from Chapter 4 provides the foundation for supporting Sternberg's six metacognitive processes, including: define the nature of the problem, generate possible steps needed to solve the problem, order the steps, decide how to represent information about the problem, allocate mental and physical resources to solve it, and monitor the solution. It also provides the foundation for researching some schematic operations, such as match, invoke, read, order, and check, proposed in Table 3.1. At this time, this research reliably supports 6 processes from metacognition and 6 strategies from schemata that are used to write a computer program (see Table 5.1). Provided in the taxonomy in Table 5.1 is a brief definition (see Appendix I for a longer definition) and an example of each metacognitive process and schematic strategy, which are high-level schemata repre-



Processes/Strategies	Brief Definition	Example
Metacognition		
Start/Revisit Goals	Explicitly or implicitly beginning or returning to tasks and subtasks in the problem statement	"Ok, now I need to go back to the loop I was working on."
Understand Problem/Plan	Thinking about the problem statement, reading/ re-reading the problem, or scheduling time/activities.	"First I have to sum the elements in z, then I can take the square root."
Read/Consider Design	Reading information from the piece of paper containing his/her design, drawings, and/or notes	"Let me look at the design to make sure I'm right."
Verbalize: Low Prior Knowledge	Explicitly stating needing to learn or find out more about a particular topic, making a low confidence statement, mentioning the difficulty of a task, or guessing computer syntax and using the compiler to determine correctness	"I know I should use a for loop, but I'm not good at for loops."
Inspect	Checking code via visual/mental inspection and comparing it to own knowledge and/or using own knowledge to execute existing code mentally	"That equals one, then that will go to the second thing in the list equals one, so everything in the list will equal one and then,…"
Compare	Comparing computer code or knowledge to the problem statement	"Yep, I took the square root then printed it."
Schemata		
Design	Drawing, designing, and/or taking notes on the blank piece of paper.	"Let me sketch these ideas on paper before I begin coding."
Write Code: High/Low Prior Knowledge	Typing correct or incorrect computer program statements or mentions wanting to write a program/code	"z sub i equals x sub i plus y sub i."
Compile	Testing code via compiling it and inspecting the compiler error messages	"Now I'm going to compile and see what errors I have."
Execute	Testing code via executing a compiled program and comparing the actual and expected output or using test-write statements or a debugging application to inspect internal program data	"Good, ok, just what I wanted it to output."
Diagnose	Using reasoning strategies to identify the cause of incorrect output/error or using statements to indentify the cause of an error without explicitly stating the reasoning	"The x and y variables have the correct value, so it must be the z variable causing problems."
Fix Code	Changing code after detecting and diagnosing an error	"Ah man, I have to go fix all the loop counters."

Table 5.1: A taxonomy for the metacognitive and schematic processes used by the participants in this research to solve the problem in Appendix E.

senting strategies from procedural knowledge that determine the context for using other schemata. The domain-specific strategies identified in this research were derived from the waterfall and iterative methods for software development, i.e. requirements/plan, design, implementation, testing, evaluation, and maintenance. For example, *write code* is a domain-specific strategy used in computer science that determines the syntax of the conceptual and procedural knowledge, which are represented by schemata, needed for implementation. The results of this research address the existence of only these schemata, their interactions with one another, and the interactions with the 6 metacognitive processes identified in the taxonomy. Future studies can determine how these high-level schemata interact with other computer science strategies, procedures, concepts, facts, experiences, and design patterns.

To address this research hypothesis on the existence of metacognition, schemata, external sources of information, and their interactions, the final taxonomy in Appendix I was used to code Level 1 columns G through K for each participant (see Appendix G). The Level 1 coding translated into problem behavior graphs, which are concise descriptions of the state of knowledge and the operations used to transfer one state of knowledge to another while solving a specific problem [146], consolidating 200 pages from 10 participants' Level 1 coding into 58 pages of graphs. A hard-copy example of one problem behavior graph can be seen in Appendix O. Initially, the Level 2 coding for the first two participants averaged 1 hour for every 6 minutes of participation, but this time was reduced to 1 hour for every 8 minutes for the next three. The Level 2 coding for the last five participants was further reduced to 1 hour for every 10 minutes. These problem behavior graphs capture the participant's movement through the goals in the problem; the metacognitive and schematic processes used within each goal; and the misconceptions, which implies errors in one's conceptual knowledge, as well as other errors occurring in metacognition and schemata as defined in Section 3.3.

5.2 The Existence of Metacognition and Schemata

In order to support the need for metacognition, schemata, and external sources of information in computer science problem solving, the sum of visits to each process and external source identified in the Level 2 coding was calculated for each participant (see Table 5.2). Based on individual participants, Table 5.2 addresses the total visits to the 6 metacognitive processes and 6 schematic strategies from the taxonomy in Table 5.1 as well as to external sources. Since the hypothesis for this research addresses the establishment of metacognitive processes, schemata, and external sources in computer science problem solving as well as the interactions among them and external sources (see Figure 3.2), the results presented here do not address the time spent within each process, even though this information is recoverable from the data (see Chapter 6.2). Table 5.2 supports the existence of metacognition and schemata in computer science problem solving based on these participants' uses of metacognitive processes ranging from 27% to 54% of their total mental processes, which exclude their uses of external sources, with an average of 44%, and 80% of these participants used metacognition in 40-50% of their mental processes. These numbers support metacognition as a tool for problem solving, and 90% of the participants had to use the Internet, books, or the experimenter to solve the problem.

Processes Visited	**Participant**									
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Metacognition	86	110	70	52	63	32	49	71	93	85
Start/Revisit Goals	28	32	15	24	17	14	16	14	24	26
Understand Problem/Plan	10	22	28	11	18	5	11	19	8	12
Read/Consider Design	0	0	0	0	0	0	0	0	0	0
Verbalize: Low Prior Knowledge	15	3	4	4	8	5	1	10	5	4
Inspect	33	50	23	11	13	7	21	26	56	42
Compare	0	3	0	2	7	1	0	2	0	1
Schemata	98	133	60	76	85	87	61	86	101	87
Design	0	1	0	0	0	0	0	0	0	0
Write Code	36	72	34	41	39	31	38	49	44	34
Compile	18	9	6	8	9	13	4	9	5	8
Execute	3	4	6	5	5	8	3	10	3	3
Diagnose	24	17	6	12	17	15	7	9	24	20
Fix Code	17	30	8	10	15	20	9	9	25	22
External Source	25	3	1	14	3	7	0	10	2	10

Table 5.2: Results for total visits to metacognition, schemata, and external sources of information per participant.

On average, participants used schemata more than metacognition or external sources, averaging 87 visits to schemata, 71 visits to metacognition, and 8 visits to external sources. Of the schemata for strategies identified in this research, *write code* was used the most, averaging 27% of all their mental processes used to solve this problem. *Diagnose* and *fix code* are the next two most used strategies in schemata averaging 9% and 10% of the total mental processes. The average visits to *fix code* exceeds the average visits to the *diagnose* strategy due to participants' lack of an implicit or explicit diagnosis. The *compile* and *execute* schemata for checking and detecting errors averaged 6% and 3% of all mental processes. The *compile* strategy is used more by participants to find the syntax errors, and after a successful compilation, the participants began using the executing strategy for finding errors. Since these participants used the *compile* strategy before using the *execute* strategy, their average use of the *compile/execute* strategy was 8% less than their use of the schema for *fix code*. This is due to the participants' use of the *inspect* metacognitive process for mentally inspecting their solution, which averaged 17% of their total uses of all mental processes.

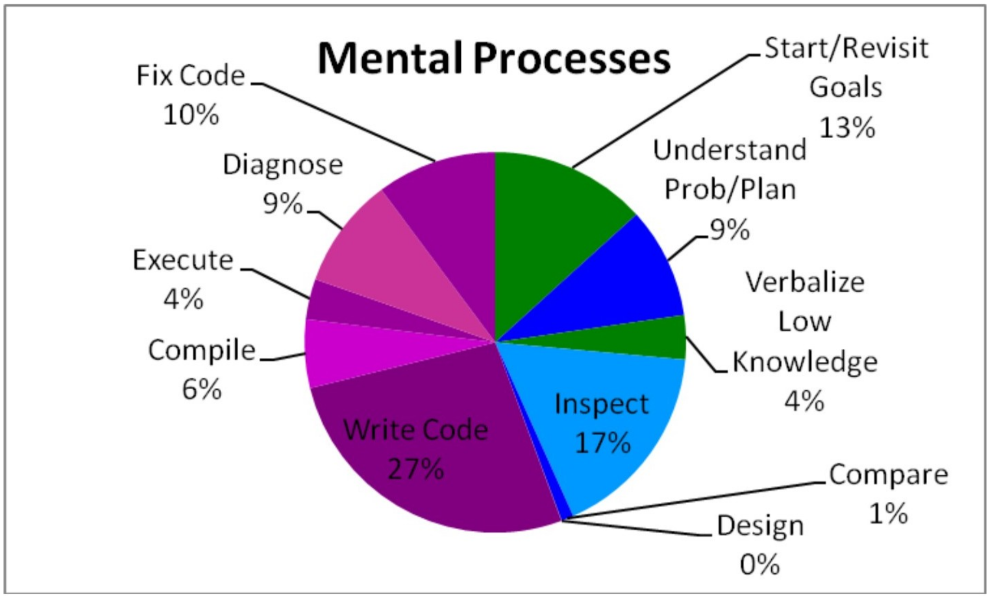


Figure 5.1: Average uses of mental processes by these 10 participants.

Metacognition was 44% of the participants' total mental processes, and the two processes, *start/revisit goals* and *inspect*, were the most widely used metacognitive processes, averaging 13% and 17% of all their mental processes used. The third most used metacognitive process was *understand problem/plan* averaging 9% of their mental processes used to solve the problem. The last metacognitive process used by every participant was *verbalize low knowledge*, which averaged 4% of their mental processes. There were two metacognitive processes not visited by all participants, and these include comparing a solution to the problem and reading/considering a design. The majority, which was 60% of the participants, compared their solution to the problem, but these visits only averaged 1% of the total mental processes. However, none of the participants used *read/consider design*, which might be attributed to 90% of the participants never using the strategy to design a solution to the problem, except for participant 3 who wrote the variable names, x, y, z, and n, on paper.

Of all the mental processes used by these participants to solve the problem in this research study, the strategy *write code* averaged the most uses, and the metacognitive process for inspecting a solution averaged 10% lower. The metacognitive processes used to *inspect* and *start/revisit goals* were used more than any other domain-specific strategy besides *write code*, and combined, these two metacognitive processes averaged more uses than any other mental process used by these participants. The fourth most used mental process was the *fix code* strategy from schemata, which averages 1% more than their visits to the *diagnose* strategy. The *understand problem/plan* and *verbalize low prior knowledge* from metacognition average more uses than the *compile* and *execute* strategies from schemata. The mental processes used the least include the metacognitive process *compare*, the *design* strategy in schemata, and the *read/consider design*. This analysis provides insight into the existence of metacognition, schemata, and external sources of information used by these participants to solve the computer science problem provided in this study.

5.3 The Relationship Between Metacognition and Schemata

The support for the existence of metacognition, schemata, and external sources in computer science problem solving allows for the analysis of the interactions among them. To capture the relationship proposed in Figure 3.2, the Level 2 problem behavior graphs were color-coded and grouped by location, and similar processes were combined based on functionality (see Figures 5.2 to 5.26). The *design* strategy and metacognitive process *read/consider design* were dropped from these graphs due to the lack of use, and the *inspect* and *compare* processes from metacognition, as well as the *compile* and *execute* strategies from schemata, were combined. Both processes and strategies are used for checking and detecting errors, but each set resides in different areas of the model. On average, participants metacognitively compared their solution to the problem statement 1% of all their mental processes, and the *execute* strategy was not used without being proceeded by a successful compilation of the program. The *start/revisit goals* and *verbalize low prior knowledge* metacognitive processes are grouped together and colored blue to classify them as control, which is a type of metacognitive process responsible for choosing and invoking specific schemata as well as determining when to visit an external source; whereas, the *understand problem/plan* and *inspect/-*

compare processes from metacognition are grouped together and colored green to classify them as monitor, which is used to check the solution, detect errors, and evaluate the progress toward a goal. The visits to a strategy from schemata are colored purple and to an external source are colored orange to correspond to the colors in the original model. For individual participants, Figures 5.2 to 5.26 illustrate the relationship among the mental processes established in Table 5.1.

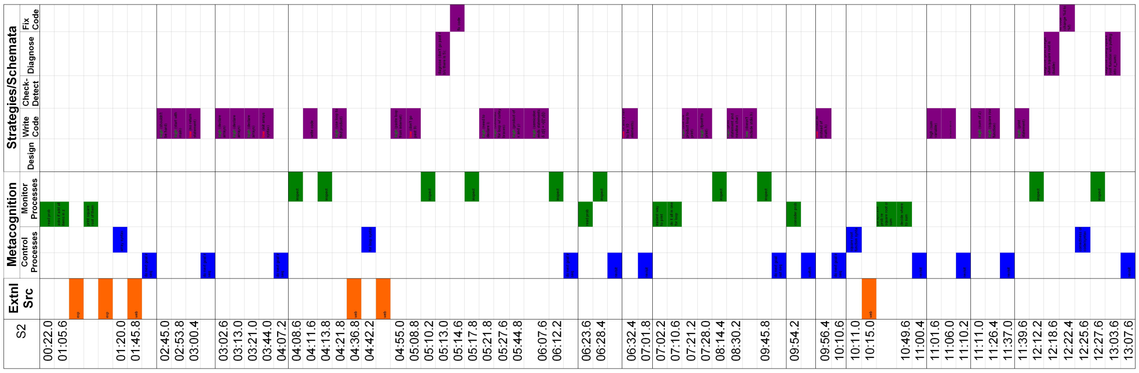


Figure 5.2: Participant 2 (0-12 minutes)

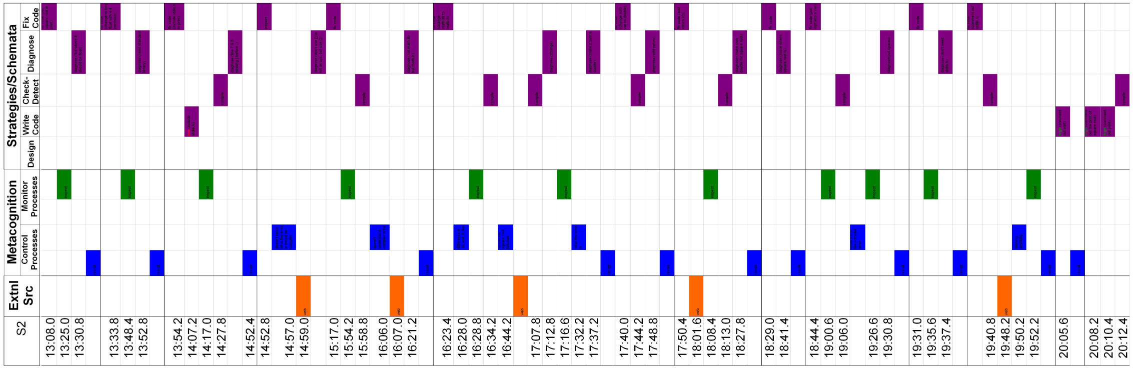


Figure 5.3: Participant 2 (12-20 minutes)

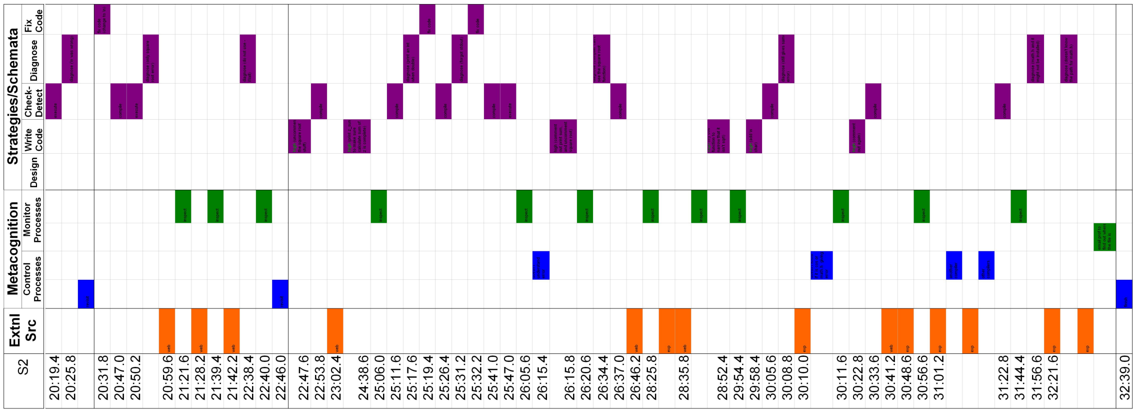


Figure 5.4: Participant 2 (20-32 minutes)

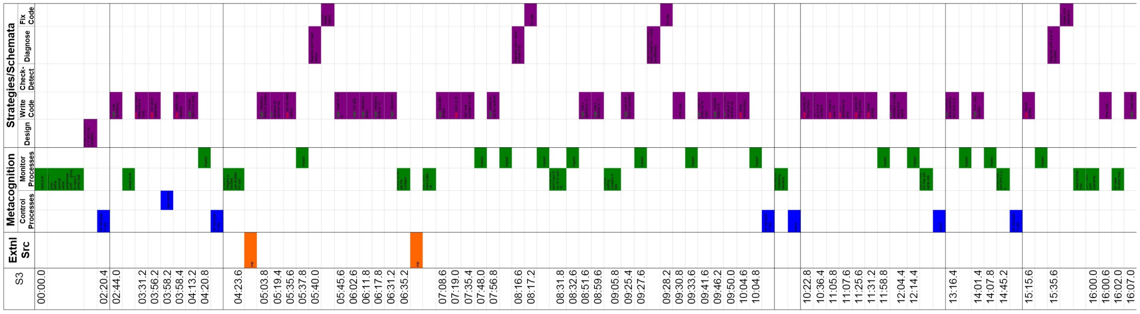


Figure 5.5: Participant 3 (0-16 minutes)

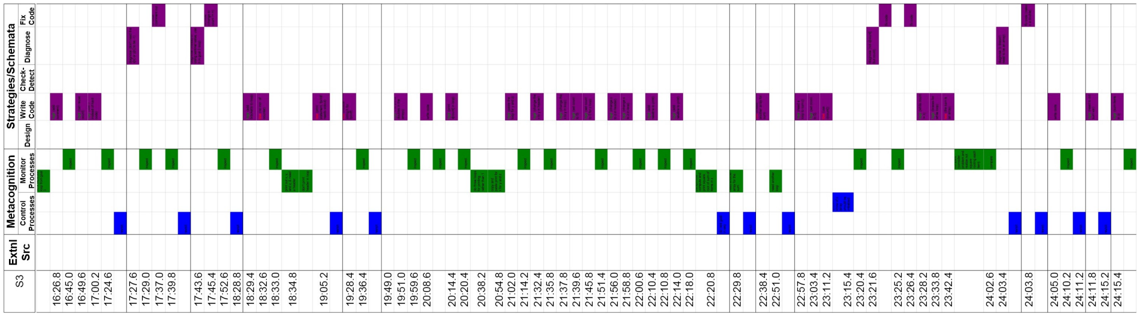


Figure 5.6: Participant 3 (16-24 minutes)

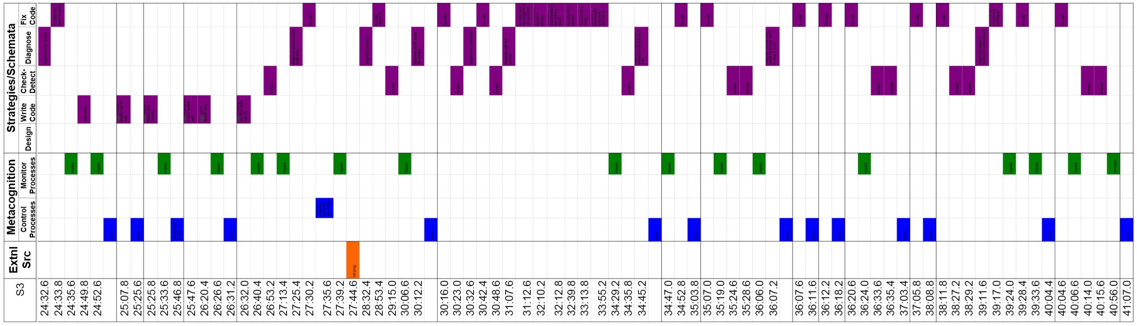


Figure 5.7: Participant 3 (24-41 minutes)

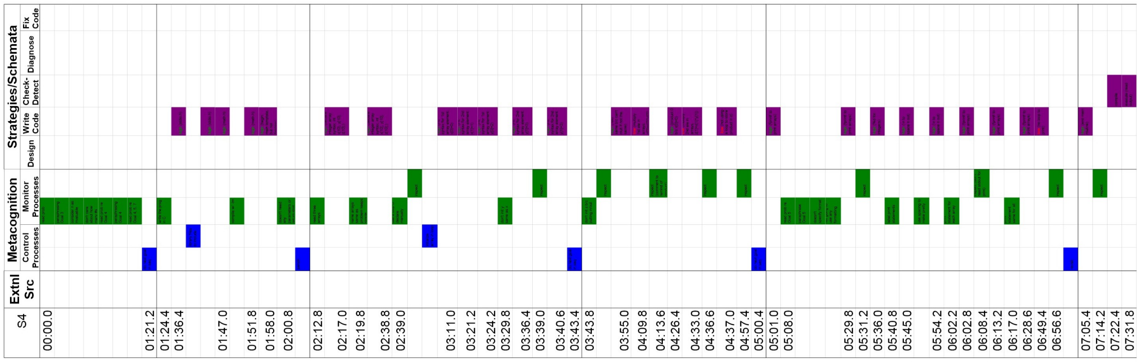


Figure 5.8: Participant 4 (0-7 minutes)

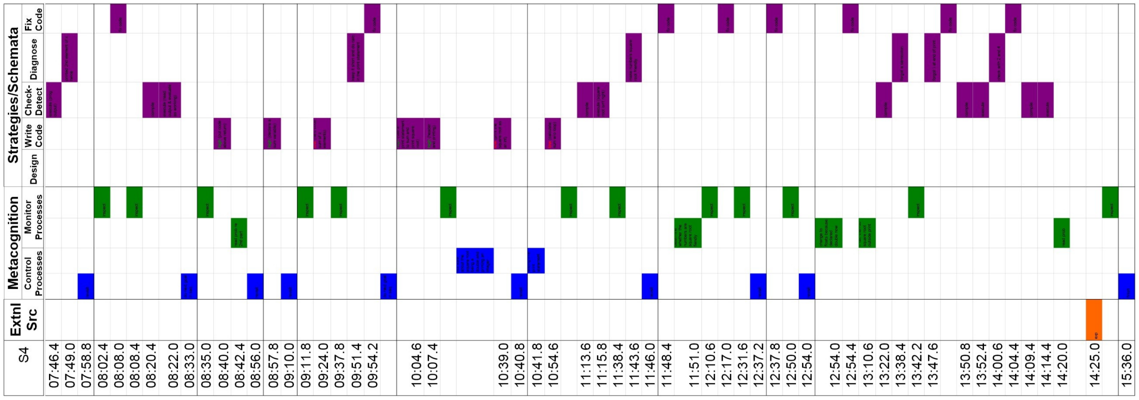


Figure 5.9: Participant 4 (7-15:30 minutes)

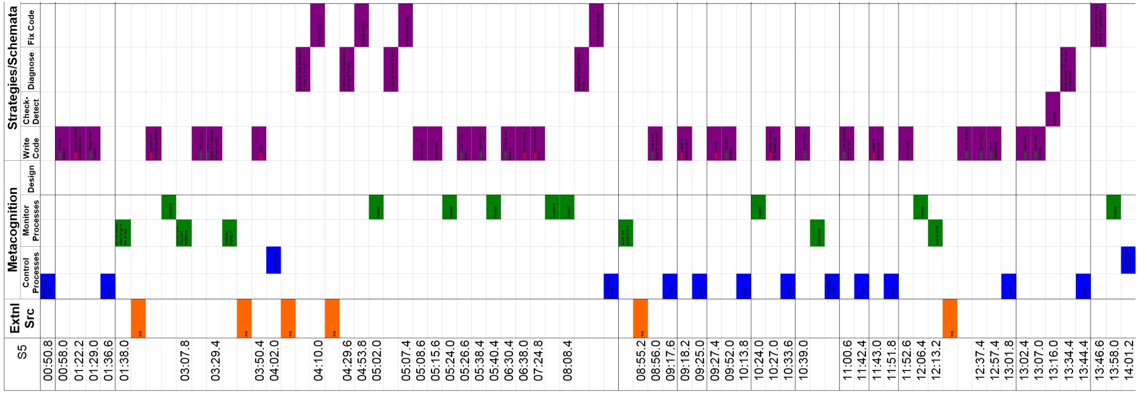


Figure 5.10: Participant 5 (0-14:45 minutes)

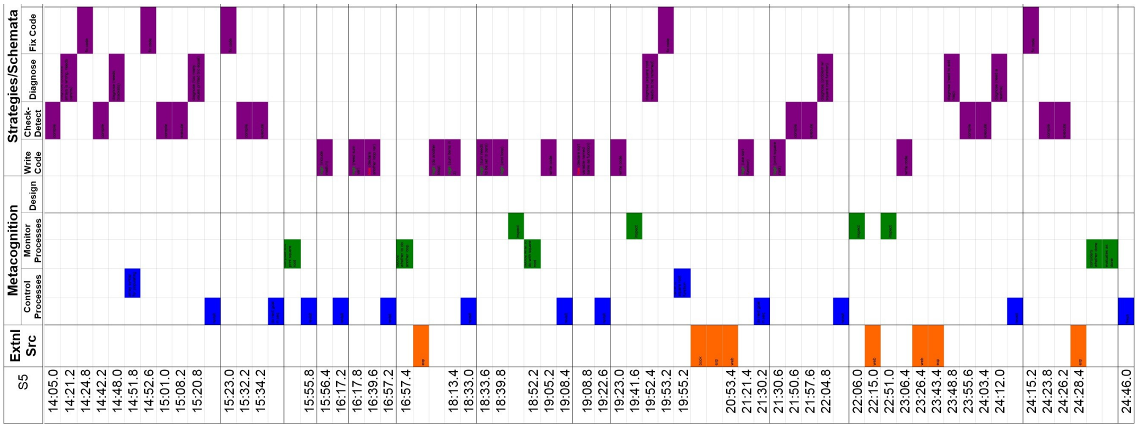


Figure 5.11: Participant 5 (14:45-24:45 minutes)

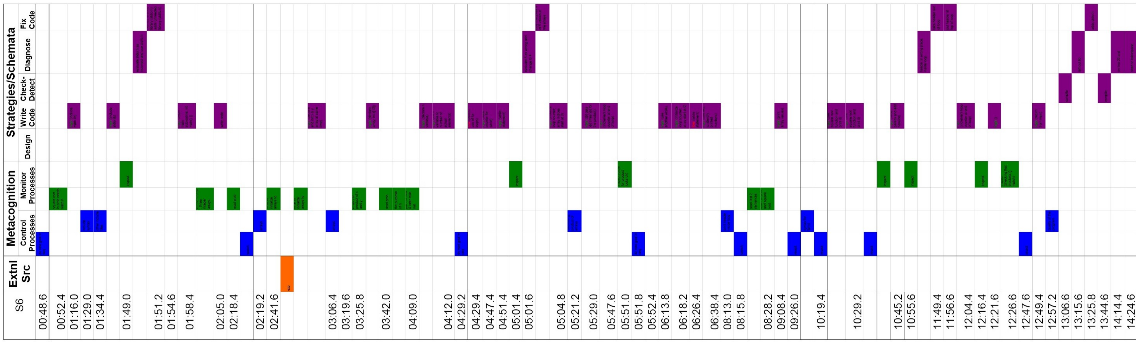


Figure 5.12: Participant 6 (0-13:30 minutes)

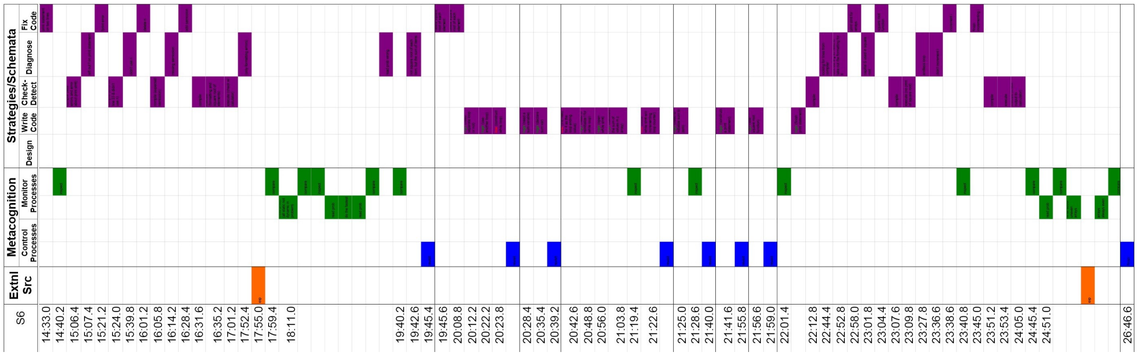


Figure 5.13: Participant 6 (13:30-26:45 minutes)

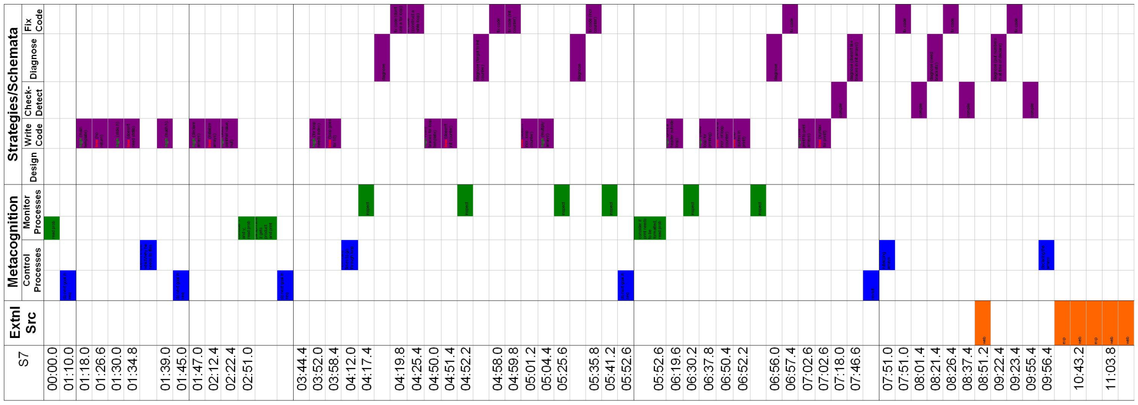


Figure 5.14: Participant 7 (0-11 minutes)

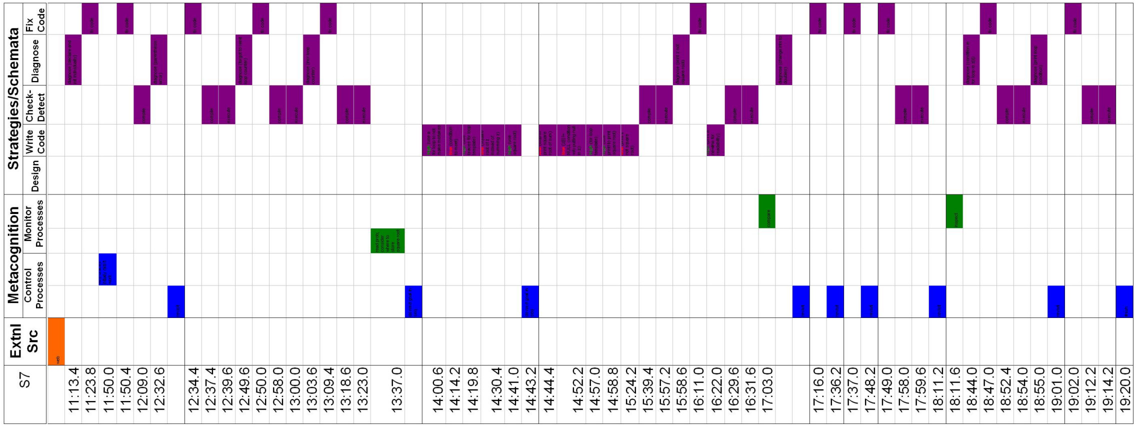


Figure 5.15: Participant 7 (11-19:20 minutes)

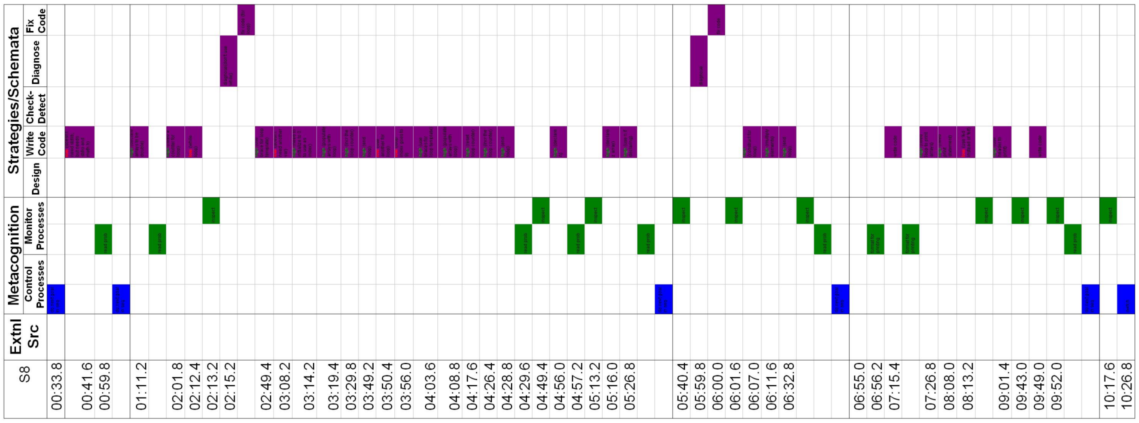


Figure 5.16: Participant 8 (0:30-10:30 minutes)

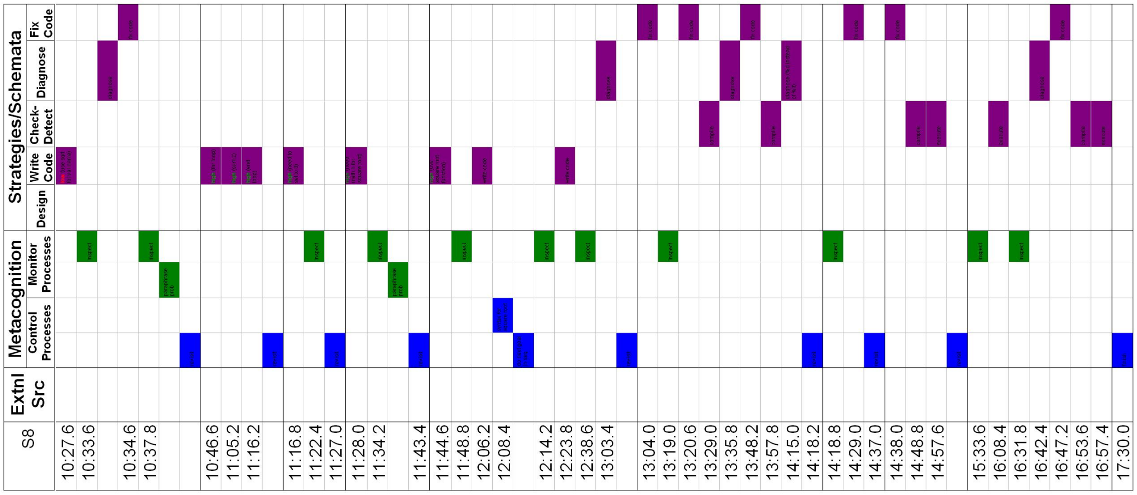


Figure 5.17: Participant 8 (10:30-17:30 minutes)

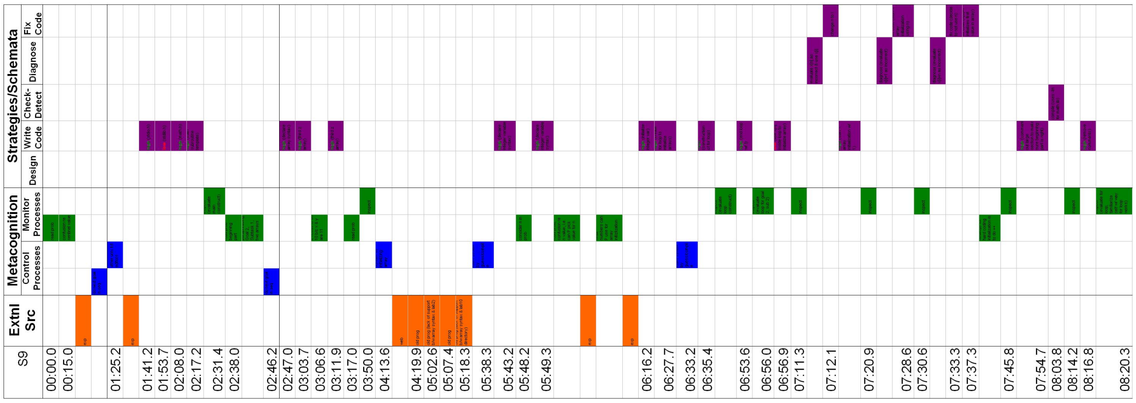


Figure 5.18: Participant 9 (0-8:20 minutes)

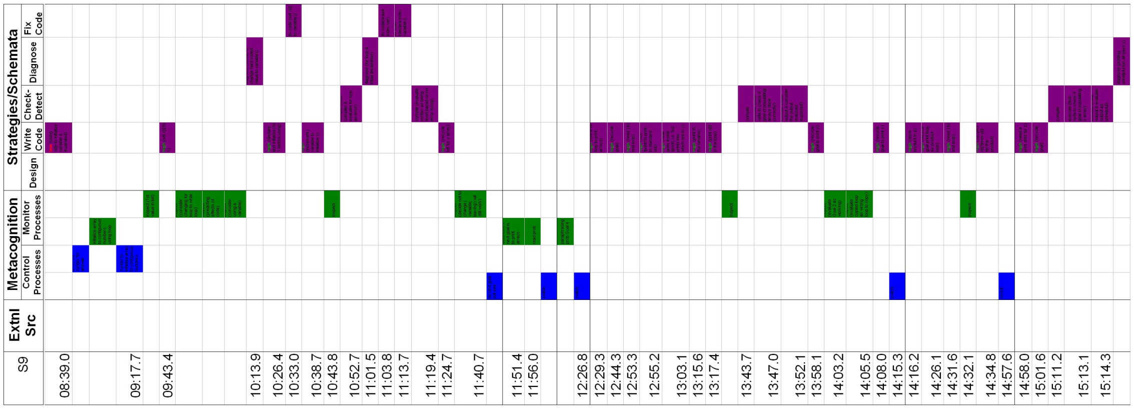


Figure 5.19: Participant 9 (8:20-15:15 minutes)

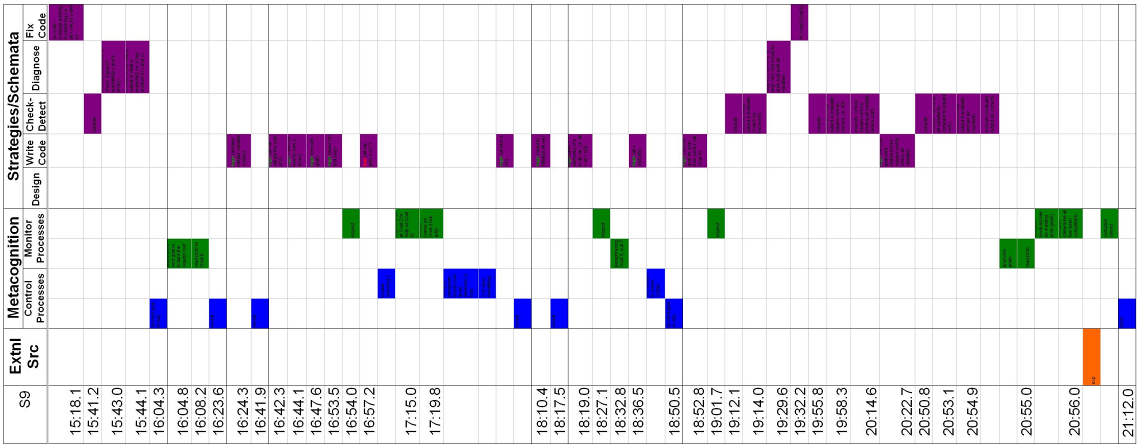


Figure 5.20: Participant 9 (15:15-21:15 minutes)

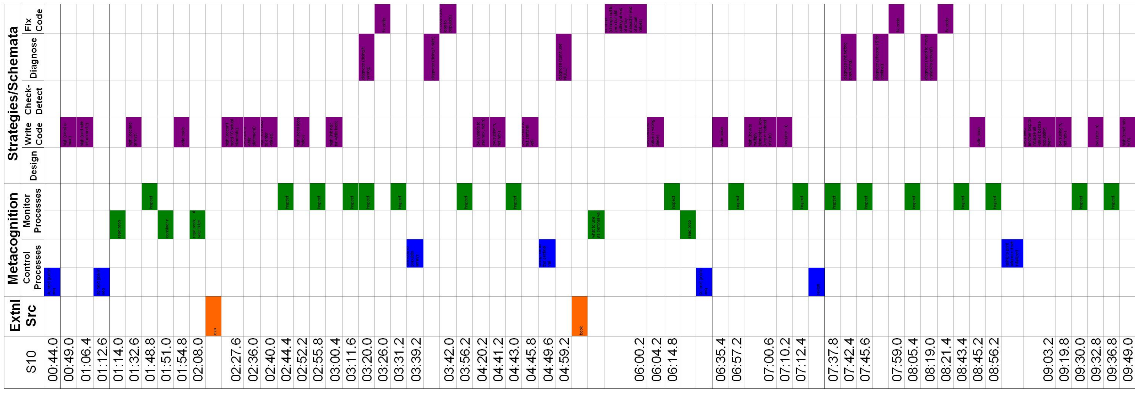


Figure 5.21: Participant 10 (0:44-9:50 minutes)

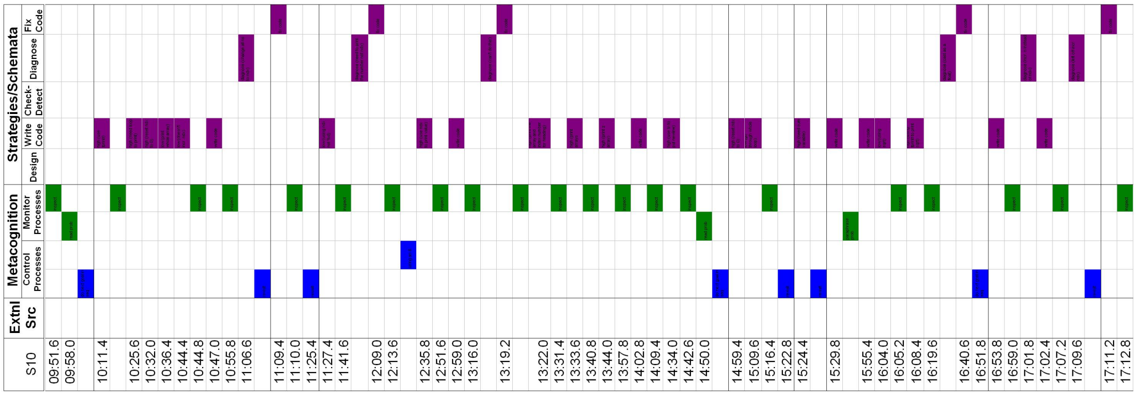


Figure 5.22: Participant 10 (9:50-17:15 minutes)

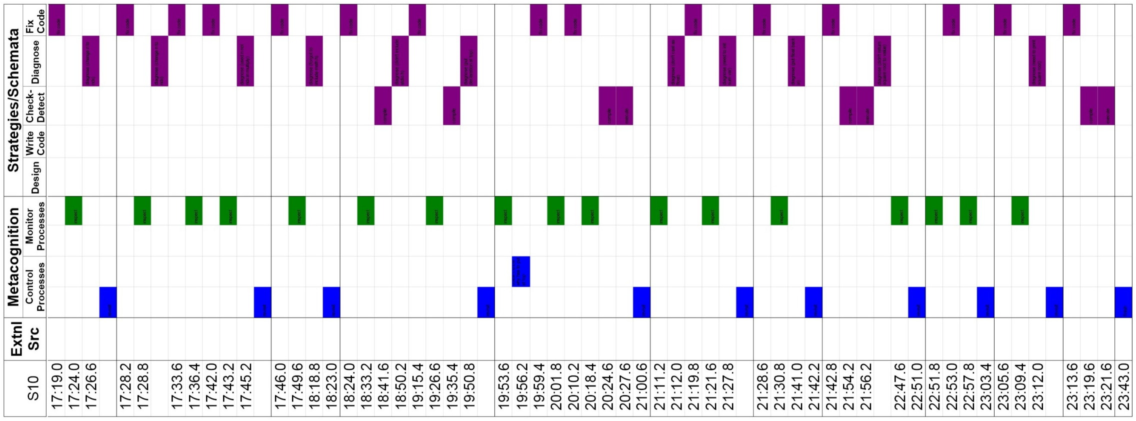


Figure 5.23: Participant 10 (17:15-23:45 minutes)

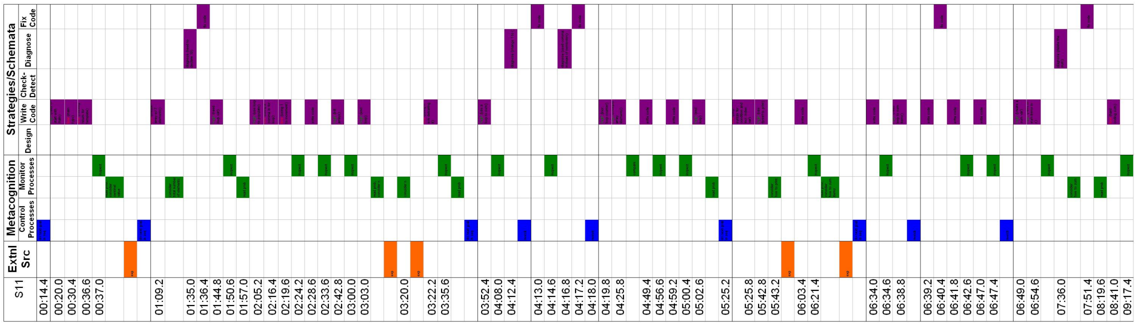


Figure 5.24: Participant 11 (0:15-9:15 minutes)

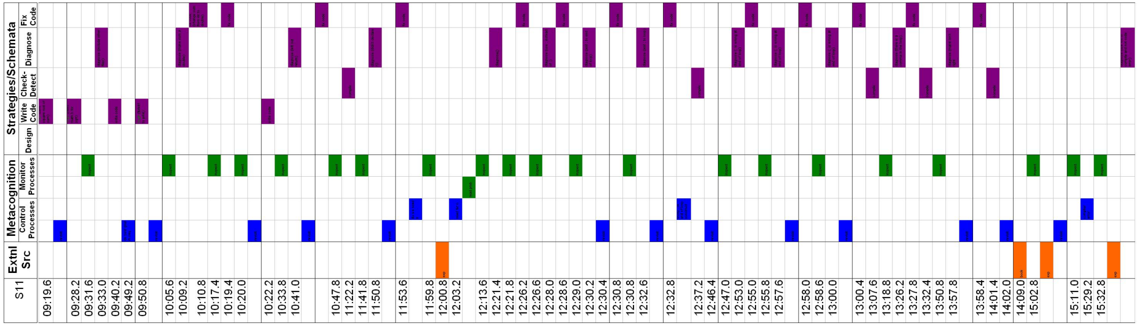


Figure 5.25: Participant 11 (9:15-15:40 minutes)

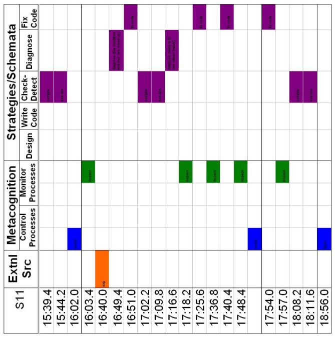


Figure 5.26: Participant 11 (15:40-19 minutes)

The graphs in Figures 5.2 through 5.26 provide a visual representation of the participants' uses of mental processes and external sources during the problem-solving exercise. The participants appear to move between control and monitor metacognitive processes, which was not proposed in the original model, and 70% of the participants, except participants 2, 5, and 7, used monitor processes more than control processes, with participants 3, 4, and 10 using them more than twice the number of those categorized as control. With the addition of the interaction between processes in metacognition and the domain-specific strategies represented by schemata, these figures support the proposed model describing how people solve problems in computer science. These additions are added to the proposed model and colored red in Figure 5.27. Even though Figures 5.2 through 5.26 support the interaction of metacognitive processes with schemata and external sources, the taxonomy used to determine these interactions does not provide enough detail to capture the operations, such as reading or writing data. Therefore, the supported model in Figure 5.27 does not distinguish the

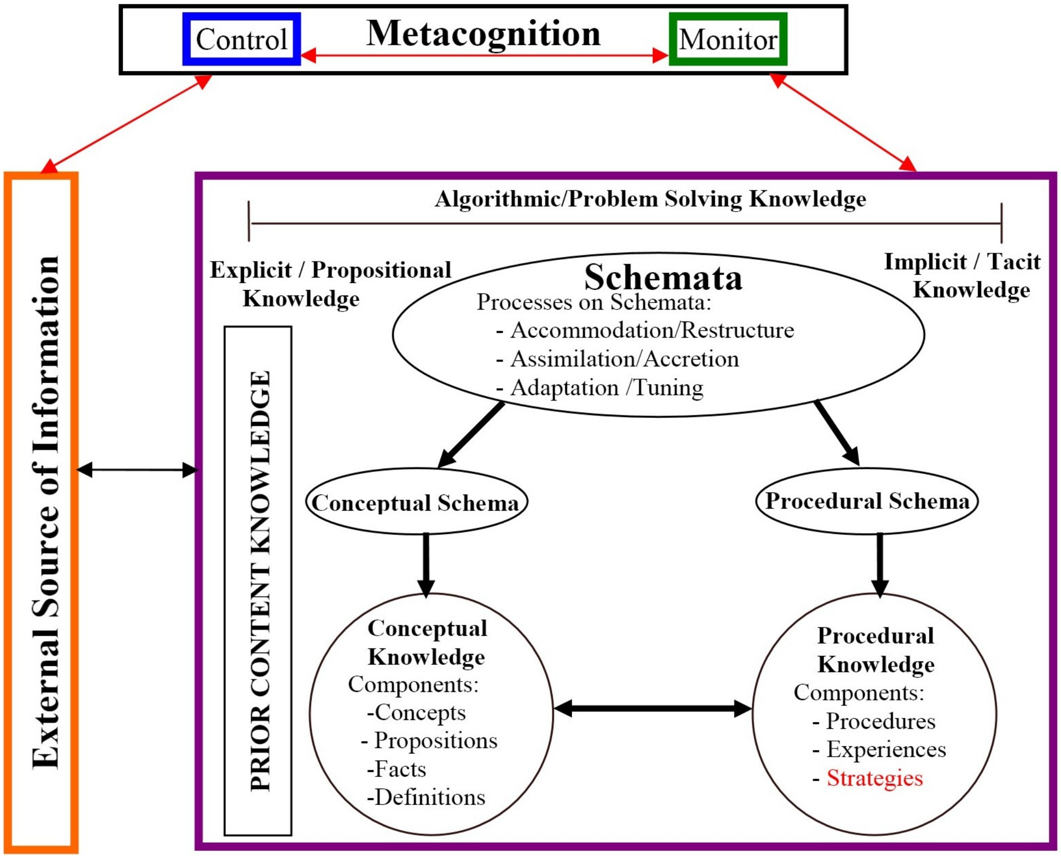


Figure 5.27: The cognitive model for cs problem solving supported by the comparison between the proposed model in Chapter 3 and the analysis of verbal protocols.

type of interactions among metacognition, schemata, and external sources. Instead, the resulting model illustrates the existence of these two-way relationships with bi-directional, unlabeled arrows. The basic terms inside the purple box representing schemata are taken from the psychology and AI literature; however, this dissertation does not investigate these terms related to computer science schemata or the labels for the arrows between schemata and its represented knowledge.

In order to analyze the interactions among specific metacognitive processes, schemata, and external sources, the interactions in Figures 5.2 through 5.26 were counted for each participant and entered into Tables 5.3 to 5.12. The design strategy was dropped from these tables, even though one participant, student 3, used it once. This participant appeared to go from *understand problem/plan* to design and from design to *start/revisit goals*, and he never used read/consider design. Tables 5.3 through 5.12 capture the participants' movements from a metacognitive process, schematic strategy, or external source to another, as well as the lack of any interactions, in order to analyze the specific interactions in the model as well as identify the cyclic behavior within the processes, strategies, and external sources. After averaging the total uses of each interaction table, the Pathfinder network analysis software [54] was used to generate knowledge networks for each participant. These participants show no interactions between a *start/revisit goals* process and another *start/revisit goals* process; whereas, all other mental processes and strategies appear to have a relationship with another mental process or strategy of the same type. For example, a participant in this study might *write code* to declare the arrays and then *write code* to initialize the arrays without using a different process or strategy. These are counted as two separate uses of the *write code* strategy for the different subgoals within the goal of creating and initializing data structures identified in Appendix E. Most interactions between the schemata for checking/detecting errors were due to a successful compilation followed by executing the code, and the other interactions were between two *execute* strategies. In this research, none of the participants appeared to move from *start/revisit goals* directly to the strategies in schemata for checking/detecting errors; whereas, 12% of the participants' uses of metacognitive processes for checking/detecting errors were directly after *start/revisit goals.* Both of the metacogntive processes and the schematic strategies for checking/detecting errors each averaged 12% of the mental processes leading to goals, but there was no evidence of participants writing code directly before diagnosing an error without using metacognition or schemata for checking/detecting errors. In addition, there was no evidence of the participants using the *compile* or *execute* strategies before fixing an error without diagnosing the error, and only 6% of the participants' uses of *fix* *code* were before fixing a different error without a diagnosis between errors. External sources were not used by these participants before fixing errors, but on average, 31% of the participant visits to external sources were before checking/detecting and diagnosing errors.

Participant #2		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal		1	3	1	10			12	
	Low Prior Knowl				5	1	1	1		7
	Understand Prob/Plan	3		3	1	1				2
	Inspect/ Compare	5	2			7	6	9		4
Schemata	Write Code	7			10	14	4			1
	Compile/ Execute		3		2		3	9		4
	Diagnose	12			2		1		5	4
	Fix Code		3		6	1	5	1		1
	External Source	1	6	3	6	2	1	4		2

Table 5.3: Participant 2's interactions among metacognition, schemata, and external sources of information.

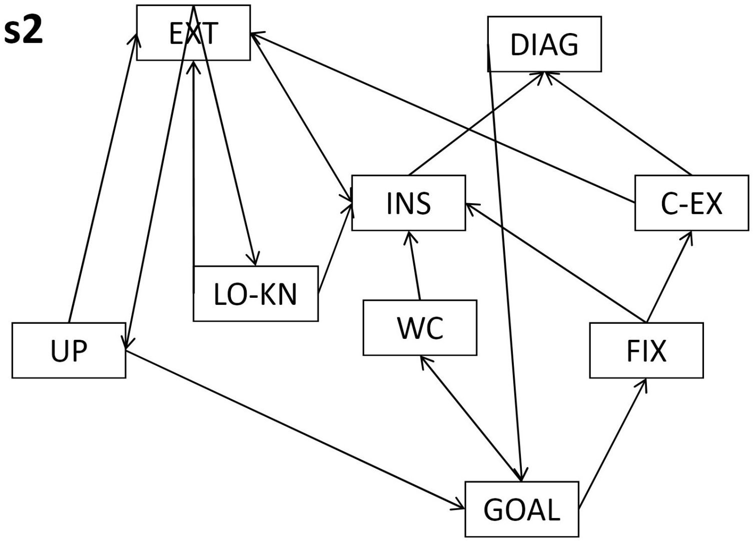


Figure 5.28: Participant 2's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #3		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal			3	1	16		2	9	
	Low Prior Knowl				2	1				
	Understand Prob/Plan	6		4	1	8				2
	Inspect/ Compare	12		6	1	14	5	10	4	1
Schemata	Write Code	3	2	6	32	29				
	Compile/ Execute	1			4		4	4		
	Diagnose	4			1				12	
	Fix Code	5	1	1	11	3	4		5	
	External Source			1		1		1		

Table 5.4: Participant 3's interactions among metacognition, schemata, and external sources of information.

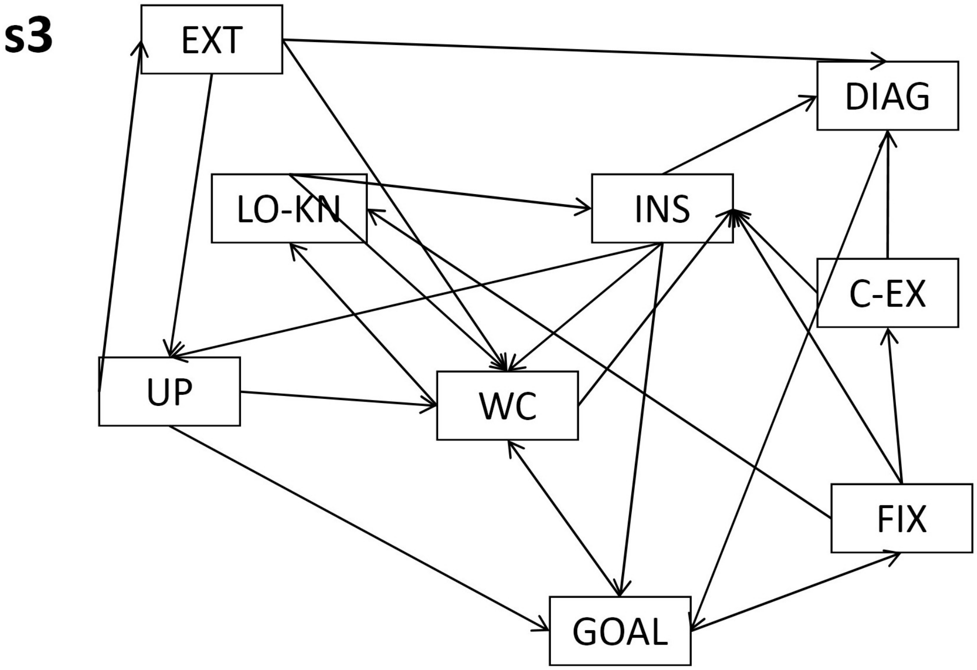


Figure 5.29: Participant 3's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #4		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal		1	4	3	4			2	
	Low Prior Knowl					4				
	Understand Prob/Plan	3		9	3	10	1		1	1
	Inspect/ Compare	5	2			8	3	3	2	
Schemata	Write Code	3	1	11	11	8				
	Compile/ Execute	1		1	1		6	3		
	Diagnose	2			1				3	
	Fix Code	1		2	3		2			
	External Source				1					

Table 5.5: Participant 4's interactions among metacognition, schemata, and external sources of information.

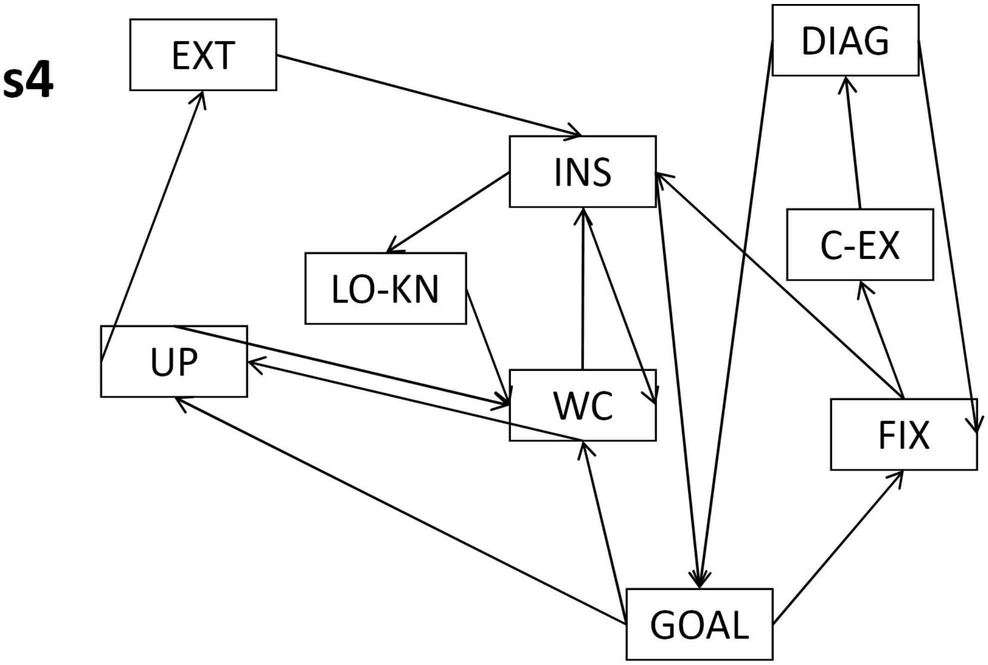


Figure 5.30: Participant 4's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #5		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal			4	2	14			3	
	Low Prior Knowl						1		1	2
	Understand Prob/Plan	3		1		2				5
	Inspect/ Compare		1	3	1	4		3		1
Schemata	Write Code	14	1	2	7	14	2			1
	Compile/ Execute	1					5	6		1
	Diagnose	4	1				1		6	
	Fix Code	1	1		2	1	4			1
	External Source			1	1	6		3		3

Table 5.6: Participant 5's interactions among metacognition, schemata, and external sources of information.

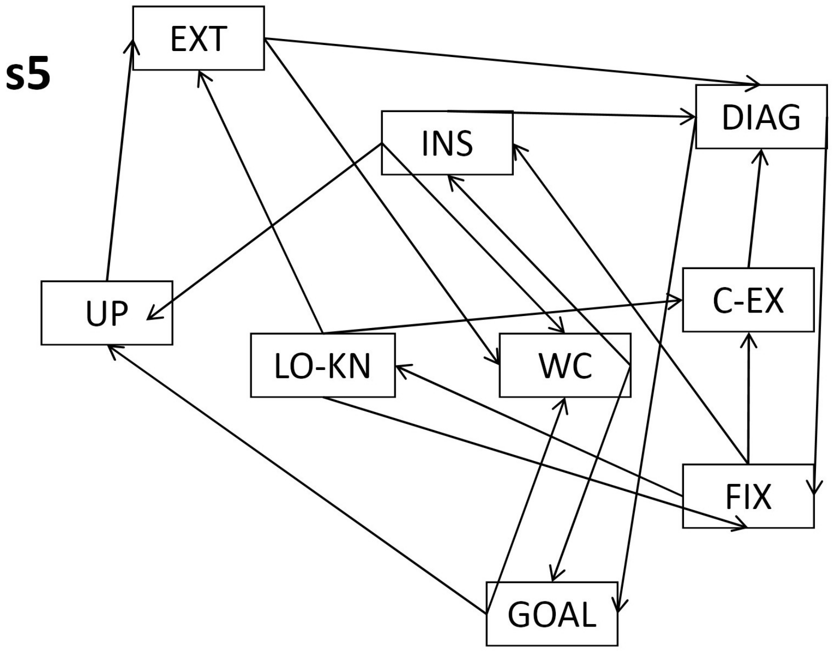


Figure 5.31: Participant 5's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #6		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal		2	2	2	9			1	
	Low Prior Knowl	2	1	2		2	1			
	Understand Prob/Plan	1		5	4	6				2
	Inspect/ Compare	4		4	1	4	1	5	1	
Schemata	Write Code	8	5	3	8	14	1			
	Compile/ Execute				1		5	8		
	Diagnose	1			1			3	11	1
	Fix Code				2	4	6	1	2	
	External Source			2	1					

Table 5.7: Participant 6's interactions among metacognition, schemata, and external sources of information.

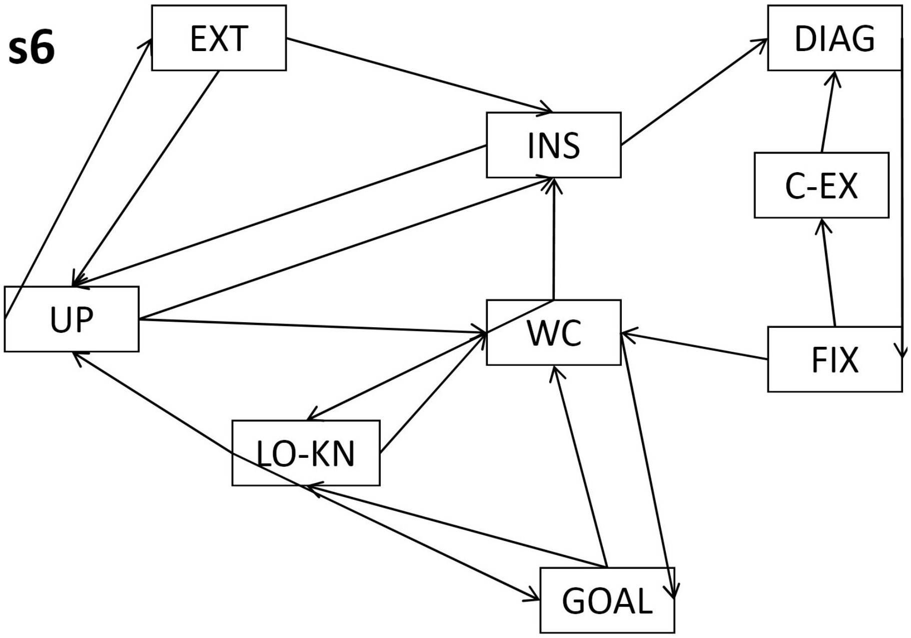


Figure 5.32: Participant 6's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #7		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal		1	1	1	5			5	
	Low Prior Knowl				1	1			2	1
	Understand Prob/Plan	3		1		1				
	Inspect/ Compare	1				1		6		
Schemata	Write Code	2	2	1	4	19	3			
	Compile/ Execute	2	1	1	1		8	7		1
	Diagnose	4							11	
	Fix Code	2	1		1	4	10		2	
	External Source							2		5

Table 5.8: Participant 7's interactions among metacognition, schemata, and external sources of information.

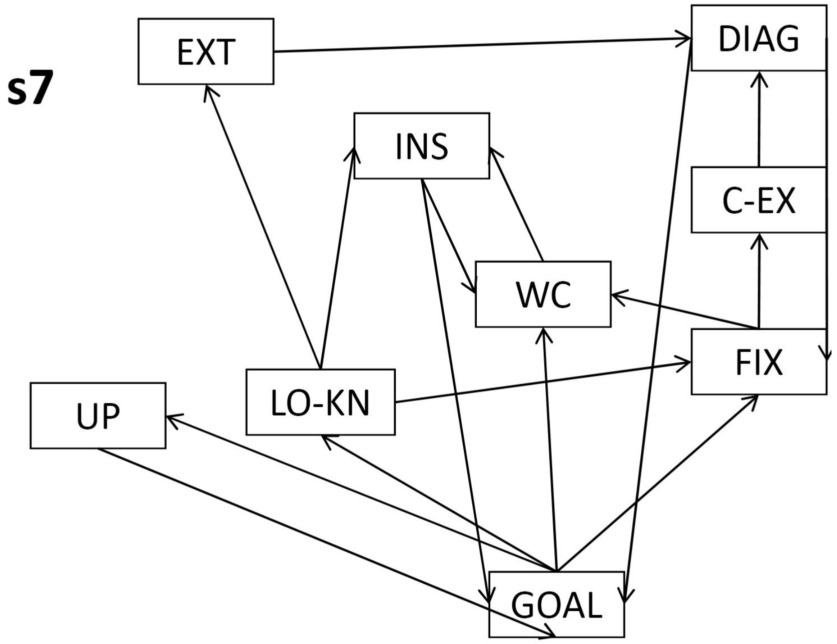


Figure 5.33: Participant 7's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #8		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal			1	5	7			2	
	Low Prior Knowl	1								
	Understand Prob/Plan	6			2	3				
	Inspect/ Compare	2		4		7	1	5	2	
Schemata	Write Code	1	1	6	10	20				
	Compile/ Execute	2			1		2	2		
	Diagnose	2							5	
	Fix Code	1			3	1	4			
	External Source									

Table 5.9: Participant 8's interactions among metacognition, schemata, and external sources of information.

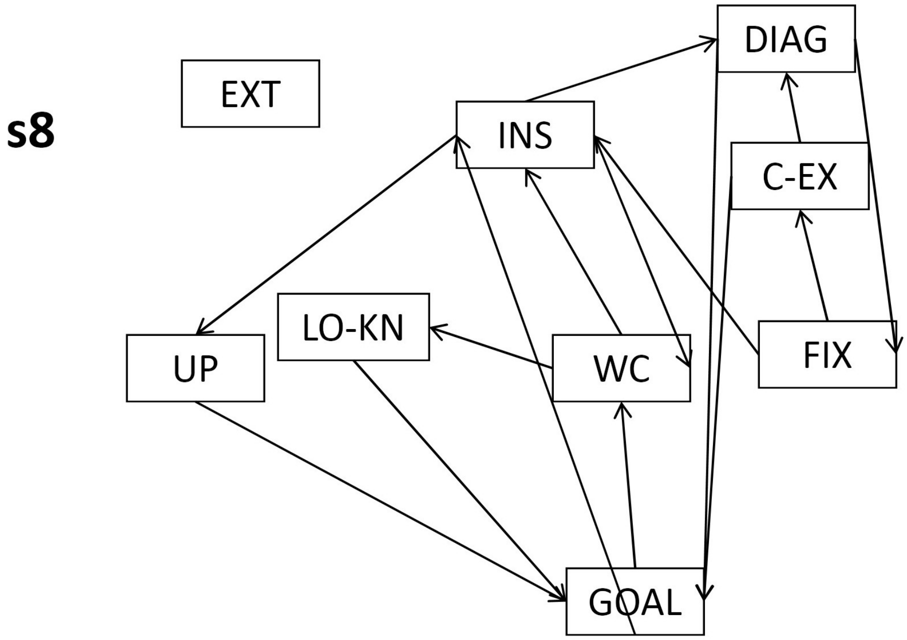
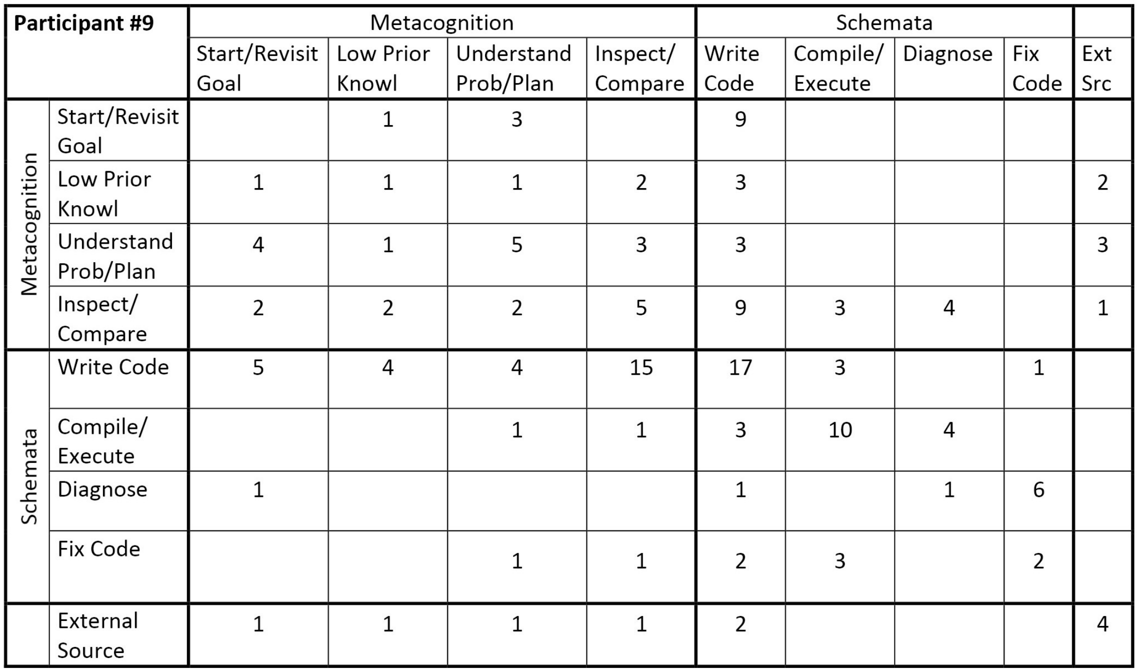


Figure 5.34: Participant 8's Pathfinder network for uses of metacognition, schemata, and external sources of information.



Participant #9		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal		1	3		9				
	Low Prior Knowl	1	1	1	2	3				2
	Understand Prob/Plan	4	1	5	3	3				3
	Inspect/ Compare	2	2	2	5	9	3	4		1
Schemata	Write Code	5	4	4	15	17	3		1	
	Compile/ Execute			1	1	3	10	4		
	Diagnose	1				1		1	6	
	Fix Code			1	1	2	3		2	
	External Source	1	1	1	1	2				4

Table 5.10: Participant 9's interactions among metacognition, schemata, and external sources of information.

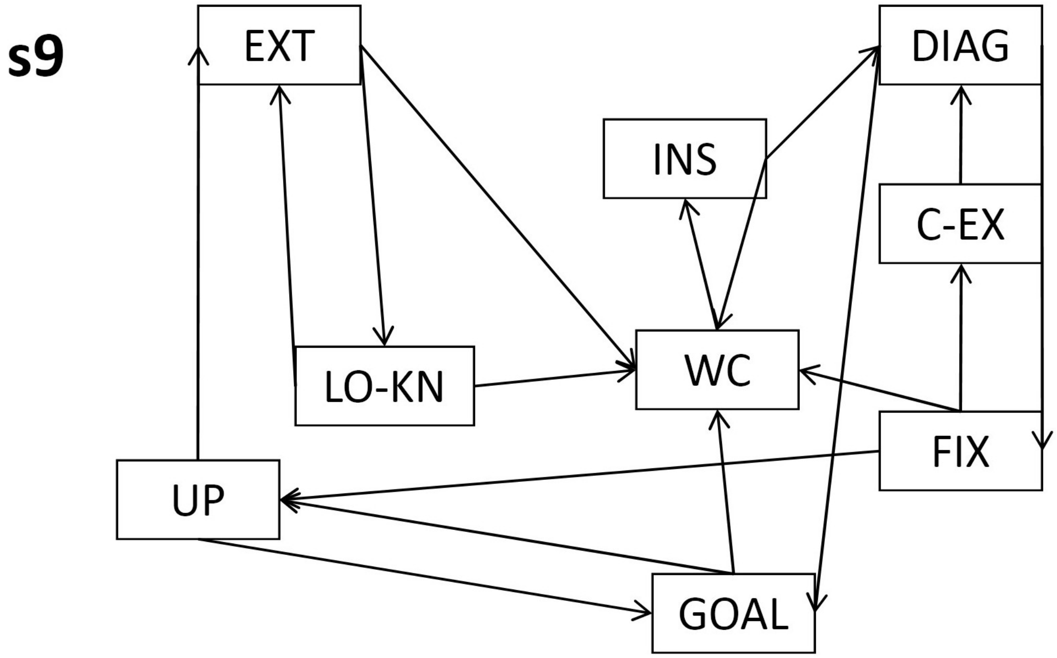


Figure 5.35: Participant 9's Pathfinder network for uses of metacognition, schemata, and external sources of information.

Participant #10		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal			1	4	8			10	
	Low Prior Knowl					2		2	1	
	Understand Prob/Plan	3				3			1	1
	Inspect/ Compare	5	4	4	1	17	3	19	3	
Schemata	Write Code	2	1	2	28	11				
	Compile/ Execute	2					3	3		
	Diagnose	10			2	1			10	1
	Fix Code	1			21	1	2			
	External Source			1		1				

Table 5.11: Participant 10's interactions among metacognition, schemata, and external sources of information.

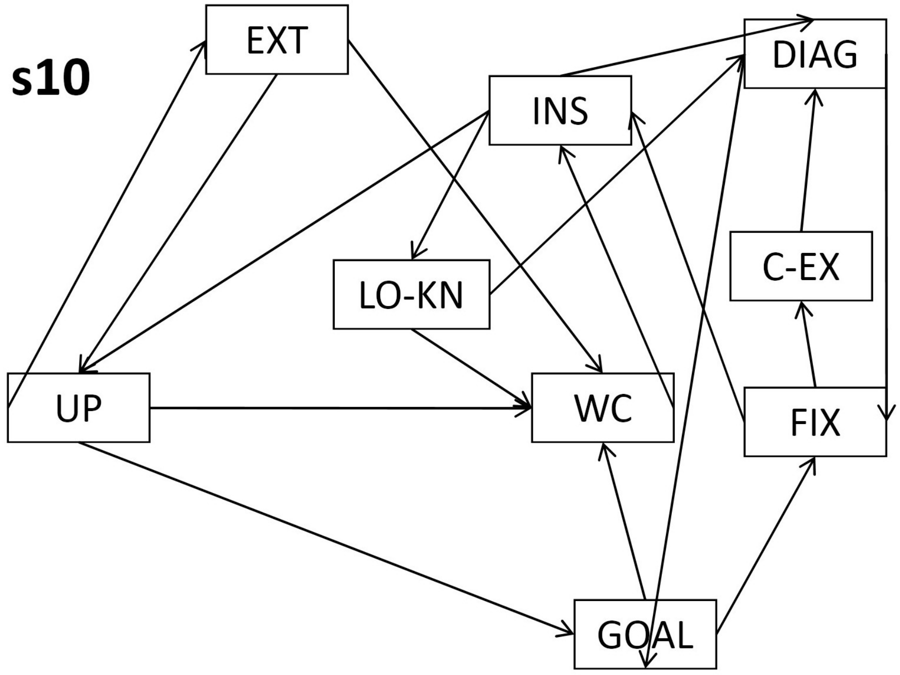
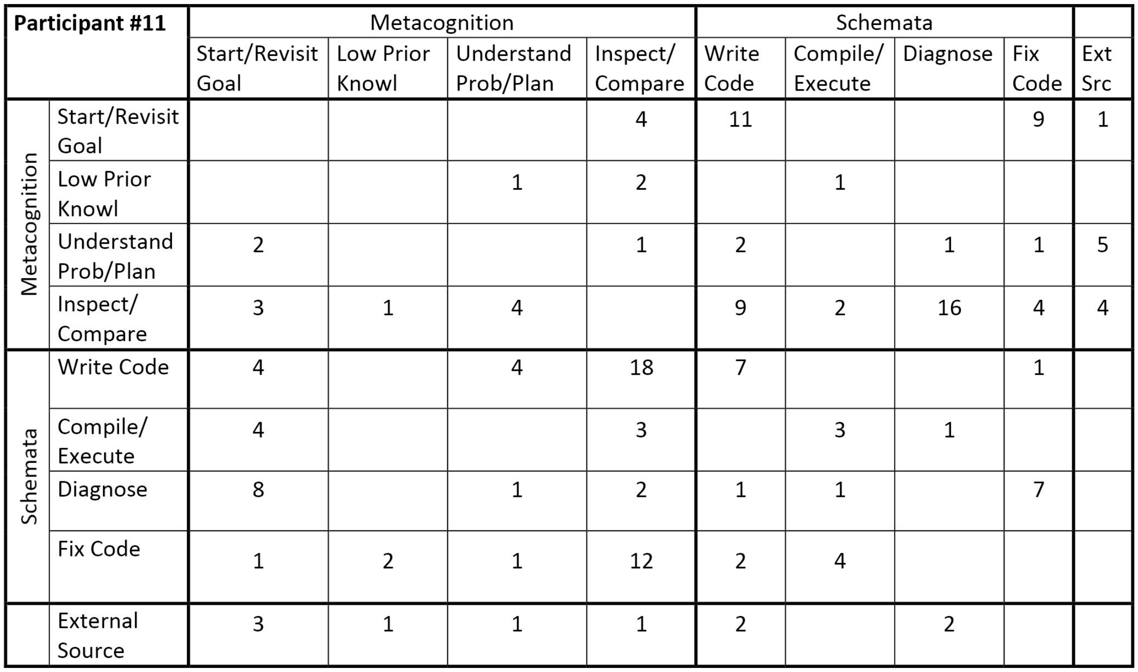


Figure 5.36: Participant 10's Pathfinder network for uses of metacognition, schemata, and external sources of information.



Participant #11		Metacognition				Schemata				
		Start/Revisit Goal	Low Prior Knowl	Understand Prob/Plan	Inspect/ Compare	Write Code	Compile/ Execute	Diagnose	Fix Code	Ext Src
Metacognition	Start/Revisit Goal				4	11			9	1
	Low Prior Knowl			1	2		1			
	Understand Prob/Plan	2			1	2		1	1	5
	Inspect/ Compare	3	1	4		9	2	16	4	4
Schemata	Write Code	4		4	18	7			1	
	Compile/ Execute	4			3		3	1		
	Diagnose	8		1	2	1	1		7	
	Fix Code	1	2	1	12	2	4			
	External Source	3	1	1	1	2		2		

Table 5.12: Participant 11's interactions among metacognition, schemata, and external sources of information.

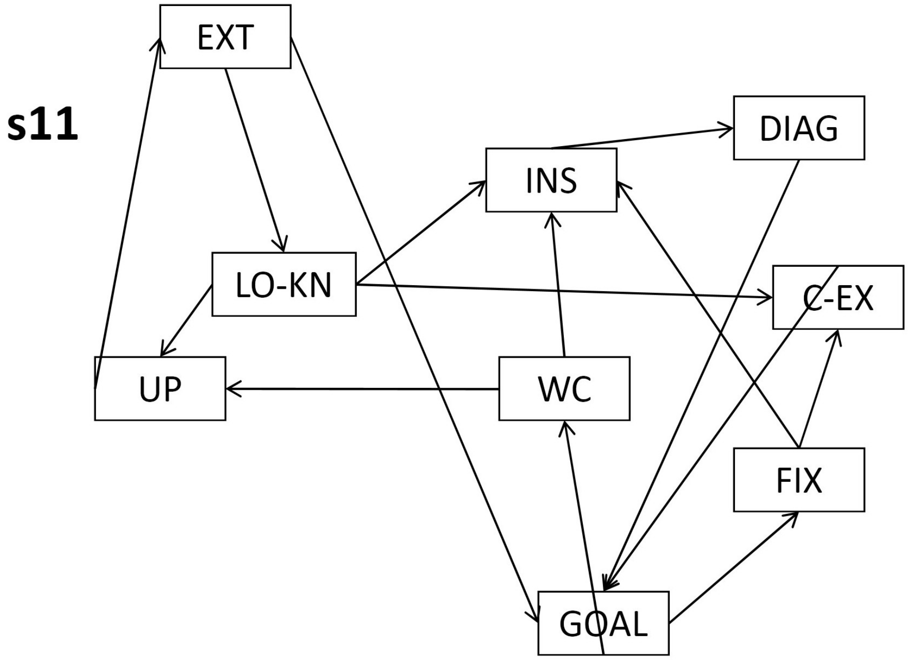


Figure 5.37: Participant 11's Pathfinder network for uses of metacognition, schemata, and external sources of information.

While there were only a few processes and strategies from metacognition and schemata showing no interactions, the processes and strategies listed below had less than 5% of their interactions with other metacognitive processes or schemata. These interactions include:

- starting/revisiting goals before verbalizing low prior knowledge or diagnosing an error
- verbalizing low prior knowledge before diagnosing an error
- understanding the problem/plan before verbalizing low prior knowledge, compiling/executing code, diagnosing an error, or fixing an error
- inspecting/comparing the solution before verbalizing low knowledge or fixing an error
- writing code before verbalizing low knowledge, compiling/executing code, or fixing an error
- compiling/executing code before verbalizing low knowledge, understanding the problem/planning, or writing code
- diagnosing an error before verbalizing low knowledge, understanding the problem/planning, or compiling/executing code
- fixing an error before verbalizing low knowledge, understanding the problem/planning, or diagnosing an error

This list of one-way relationships averaging less than 5%, with some less than %1, plus the list of no interactions provide little support for the interactions between the *write code* and *diagnose* strategies, the *write code* and *compile/execute* strategies, the *diagnose* strategy and metacognitive process for verbalizing low knowledge, and the *fix code* strategy and the *understand problem/plan* process from metacognition. In addition, the metacognitive processes *verbalize low knowledge* and *inspect/compare* as well as the *diagnose* strategy from schemata had less than a 5% relationship with another process or strategy of the same type, and none of the strategies from schemata averaged greater than 3% of their uses leading to an external source. This low interaction implies that these participants used metacognitive processes to determine when, what, and where to visit an external source. On average, 30% of the visits to two metacognitive processes, *verbalize low knowledge* and *understand problem/plan*, were used before visiting an external source, and 14% of the visits to external sources resulted in the use of another external source without visiting metacognition or schemata between sources. It appears that 66% of the participants' uses of external sources were to *understand problem/plan*, *inspect/compare*, *write code*, or *diagnose* an error, while other interactions averaged less than 5% and 1%.

Even though the interactions between metacognition, schemata, and external sources listed above resulted in less than 5% of their interactions with other processes and strategies in this study, there are specific processes and strategies from metacognition and schemata that averaged higher than 10%. The use of the *diagnose* strategy before fixing an error was the only interaction with an average greater than 50%, and 30% of the other uses of *diagnose* were to revisit goals implying that the participants diagnosed an error outside their current goal requiring revisiting a prior goal before fixing the error. This interaction between *diagnose* and *start/revisit goals* explains why 24% of the visits to goals were directly before fixing an error. Participant 9 was the only one who did not have any interactions between *start/revisit goals* and *fix code*, which implies that most of his errors were detected and corrected while working on a goal before moving to the next goal. The majority, 47%, of the uses of the metacognitive process for *start/revisit goals* resulted in *write code*, and another 24-25% of *start/revisit goals* interacted with a monitor process from metacognition. Even though control processes can be followed by another control process from metacognition, most had interactions with schemata and external sources, and more specifically, 39% of the verbalizations of low knowledge led directly to *write code* or *start/revisit goals* without using an external source to correct the low knowledge. However, 14% of their verbalizations of low knowledge did result in using an external source. It appears that the metacognitive processes classified as monitor are used before using other metacognitive processes, but the *inspect/compare* processes versus the *understand problem/plan* process average more interactions with schemata. This is due to the participants' using 29% of their mental processes to *understand problem/plan* before *start/revisit goals*, and 29% of their *inspect/compare* metacognitive processes before visiting their schemata for *diagnose*. The use of metacognition to diagnose errors was only 4% lower than the average use of the *compile/execute* strategies leading to a diagnosis, and another third of the interactions with *compile/execute* were between the compiler and executing the program. These two interactions account for 68% of all the interactions with the schemata for checking/detecting errors; while, 12% of the remaining interactions were with goals implying that participants rarely compiled or executed their program between goals without having any errors. On average, 33% of participants' uses of *write code* led to metacognitive processes for checking/detecting errors, and 25-26% of the uses of a monitor process in metacognition led back to *write code*, totaling 51% of all uses of the *write code* strategy. Another 37% of the uses of *write code* were from another *write code* strategy. Only 2% of the uses of *write code* were after using the compiler or executing the program, also showing that participants in this study rarely had successful compilations or executions and had successful ones after completing all goals, when the *write code* strategy is no longer needed. After fixing errors, participants typically used either a metacognitive process or schema for evaluating their fix and checking/detecting more errors; each averaging 33% and 30% of the mental processes after fixing an error.

The analysis of these ten participants' interactions between domain-specific strategies from schemata and the metacognitive processes reveal how students move through the software development process using metacognition to solve the problem in Appendix E (see Figure 5.38). On average, participants in this study did not use the design strategy to develop their software; therefore, showing no interactions between one of the initial software development strategies and *write code*. However, students did appear to use their metacognitive process *understand problem/plan* before *write code*. The sixth taxonomy from Chapter 4(see Appendix N) had a requirements strategy as part of its vocabulary, but the strategy was dropped due to its confusion with metacognitive planning. Figure 5.38 illustrates the appearance of these participants' relationships between their metacognition and their software development strategies occurring more than 3%. As seen in this figure, the participants do not go from *write code* to the *compile/execute* strategy, but rather, they appear to use metacognition before compiling or executing. In addition, these students' *write code* strategy had an inverse relationship with *fix code*, where an average of 12% of the uses of *fix code* returned to *write code*, and 30% of the uses returned to the *compile/execute* strategy. Even though these participants appeared to have few interactions between *write code* and *compile/execute*, they

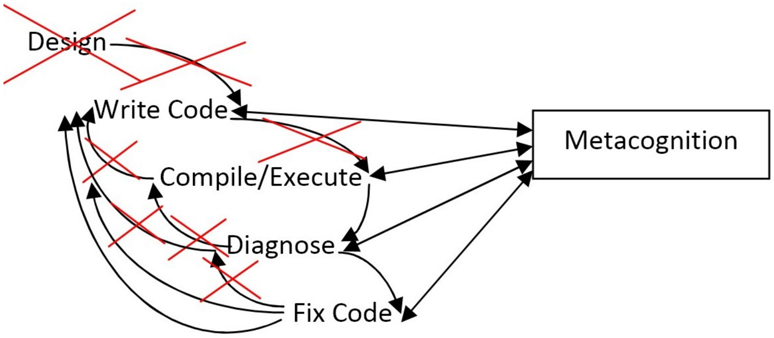


Figure 5.38: Overall interactions between domain-specific strategies derived from software development methods and metacognition.

did move from *compile/execute* to *diagnose* and from *diagnose* to *fix code* without skipping the *diagnose* strategy. However, every strategy during their software development appeared to have a relationship with metacognition, which is ignored in the current methods, and these participants used their metacognition to return to previous strategies, except when returning to *write code* or *compile/execute* from fixing an error.

5.4 Cognitive Behavior by Participant

Beyond the results supporting the existence of metacognition, schemata, and external sources and the average cognitive behavior of the participants in this study, each participant appeared to cognitively behave differently and have different knowledge represented by their schemata. Using the problem behavior graphs, participants' cognitive behavior is analyzed individually. For example, participant 3 was the only one who used the design programming strategy for this problem, and 90% of the participants, except participant 8, required outside help/hints to progress, meaning that the use of an external source is independent of the initial steps. Instead of using the design programming strategy first, all participants used their metacognition, but 50% of the participants used *understand problem/plan* before starting their initial goal, while the other 50% started a goal then went directly to *write code*. In addition, all participants ended in metacognition to determine if all goals were complete. Almost half, 40%, of the participants used schemata for checking/detecting errors before completing, while the other 60% used processes from metacognition to monitor their solution. Of these participants using metacognition before finishing, 20% used *understand problem/plan*, 30% explicitly compared their solution to the problem, and 10% visually inspected their solution to determine whether they were finished solving the problem, and two of the four participants who visited their schemata before finishing worked examples to check their answers. Unlike the small percentage of participants who used *understand problem/plan* before finishing, the majority, 70%, of the participants used this metacognitive process before *start/revisit goals*, whereas, participants 2 and 4 primarily revisited the problem after beginning a goal. Participant 9 was the only one who revisited the problem equally before and after starting a goal, and participants 3 and 8 are the only ones who exclusively who used the process before a goal. However, participant 8 was not one of the five who used *understand problem/plan* before working on the solution. Table 5.13 shows the statistics for the number of times and when each participant revisited the problem:

Problem revisited	Participant									
	2	3	4	5	6	7	8	9	10	11
Before Goal	3	4	2	2	3	3	9	4	3	5
After Goal	1	0	1	1	1	1	0	4	2	1

Table 5.13: Participant revisits to the problem before or after starting a goal.

Additional analysis of participant actions addresses the relationship between goals and error detection and correction. The most obvious difference among participants is the amount of time they checked for errors using the compiler. Three participants, 4, 7, and 9, out of ten incrementally compiled their programs as they wrote code, and two more, 5 and 6, compiled within the first half of solving the problem. Most participants diagnosed and fixed errors before compiling their programs using metacognition to inspect their solutions, but two diagnosed errors without fixing them by using the compiler to see if their diagnoses were correct. While the number of participants differs between these actions, the largest discrepancy is between the number of times the participants performed these actions. Participants 2, 5, and 11 were the only three who diagnosed errors then used the compiler for validation, performing this action 3, 2, and 1 times, respectively. However, all participants, except participant 4, diagnosed and fixed errors by inspecting their solutions before using the compiler, the number of times for each listed below:

- Participant 2 - 5 times
- Participant 3 - 9 times
- Participant 5 - 4 times
- Participant 6 - 3 times
- Participant 7 - 4 times
- Participant 8 - 4 times
- Participant 9 - 3 times
- Participant 10 - 17 times

- Participant 11 - 7 times

5.5 Knowledge Represented by Participants

In this study, the participants represented different knowledge in their solution. The problem behavior graphs provide details about the schemata used within the goals for this problem, as well as the number of participants completing each goal. Below is a list of each goal and the different schema used to accomplish it, with the number in parenthesis indicating the number of participants who accomplished the goal and who used a specific schema within each.

Schemata Details for Solution to Problem:

1. Construct a main routine (10)
    - Return 0 (8)
    - Return 1 (1)
    - No return statement (1)
2. Include library files (math.h and stdio.h) (10)
    - Included unnecessary libraries, ex. stdlib.h (6)
3. Create arrays with 10 elements (10)
4. Populate/Fill arrays (10)
    - At time of declaration (3)
    - Use loop (4)
    - Individual Elements (3)
5. Multiply two arrays and store in third (10)
    - Use while loop (4)
    - Use for loop (5)
    - Use no loop (1)
6. Print arrays (10)

   - Use while loop (4)
   - Use for loop (5)
   - Use no loop (1)

7. Sum elements in an array (10)
   - Use while loop (3)
   - Use for loop (5)
   - Use no loop (2)

8. Use square root function (10)
   - Compiled with -lm option (7)
   - Compiled with -lm using help (2)
   - Never compiled with -lm (1)

9. Use print statement to print square root (10)

As this list of goals with associated schemata and the participants' completion of each task shows, all were able to complete the goals. However, the details of the schemata used by the participants varied from incorrect to correct and inefficient to efficient. For instance, 60% of the participants included stdlib.h, which is a standard library header not needed for this solution. Even though including this file does not affect the solution, the inclusion of the header is inefficient. Seven of the ten participants initialized their arrays within a loop or at the time of declaration, whereas three inefficiently initialized individual array elements after creating the array. When populating their arrays, only 40% acknowledged the sentinel value, which was directly in the problem statement. Half of these students used a for loop to multiply and print arrays, while others used a while loop, and one used no loop at all. The shift in the number of participants using a while loop and no loops between the sixth and seventh goals is due to two students. One student switched from a while loop to a for loop after continuously forgetting to reset and increment the loop counter, and another student went from using for loops to using no loop after struggling with remembering how to sum array elements in a loop, which was a struggle to other students. All students remembered to create a sum variable, but one student forgot to initialize the variable, relying on the compiler to set the variable to zero. Seven of the ten student participants knew to compile with the -lm option, when using the gcc compiler, and two more students remembered after receiving a hint from the researcher.

5.6 Correlation With Problem-Solving Performance

To compare participant behavior with grades for the problem, two schemes for assigning grades was developed. The first such scheme assigns point values to the goals and subgoals established in Appendix E.

Grading Scheme Based on Goals

Goal 1 Begin program (10 points)

- Construct main routine +5
- Include library files +5

Goal 2 Create/Initialize data structures (30 points)

- Declare x, y, and z array with 10 elements +10
- Populate the x and y array with data values ended by sentinel value +10
- Declare other variables +5
- Initialize other variables +5

Goal 3 Compute/Store product of x and y in z (15 points)

- Construct loop to step through arrays +10
- Multiply elements in x and y and assign to z +5

Goal 4 Print arrays (15 points)

- Construct loop to step through arrays +10
- Construct print statement to output values +5

Goal 5 Compute square root of sum of z (25 points)

- Construct loop to step through arrays +10

- Calculate the sum of the values in the z array +10
- Use square root function +5

Goal 6 Print square root of sum of z (5 points)

- Construct print statement to output values +5

While this grading scheme is based only on the achievement of goals and subgoals, the other grading scheme below accounts for the quality and features of the schemata used in the solution. It uses an expanded version of the goals established by the nine associated schemata (see Appendix E).

Grading Scheme Based on Schema Details

100% - Correctly implement schemata 1-9; correctly using for loops and add/use sentinel value

90% - Implement schemata 1-9 either w/o using a sentinel value or w/o using for loops

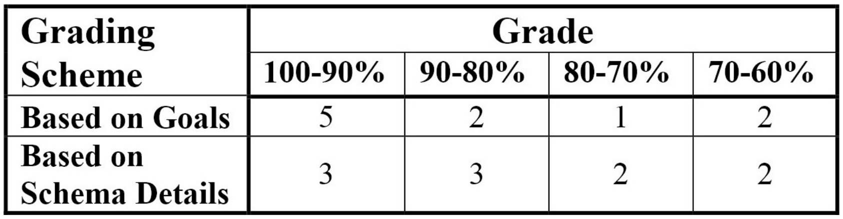
80% - Implement schemata 1-9 without using for loops and a sentinel value

70% - Implement schemata 1-9 without any loops or sentinel value

60% - Incorrect answer to problem, i.e. incorrect multiply, sum, square root, etc.

below 60% - No clue/Implementation that doesn't execute/compile

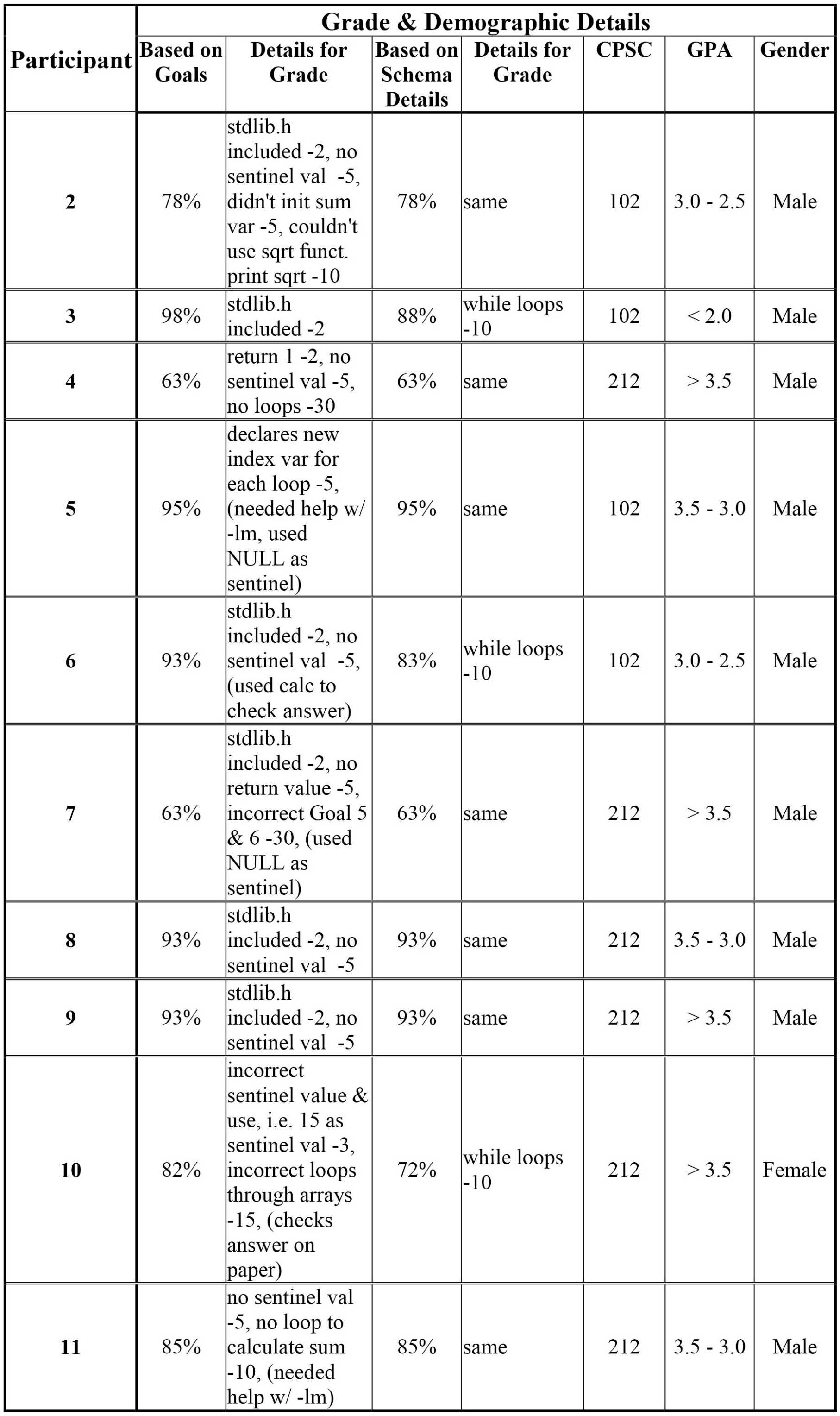
Table 5.14 provides the number of participants scoring an A, B, C, and D using the two grading schemes. As this table shows, there is a decrease in the number of As using the grading scheme based on the details of schemata.



Grading Scheme	Grade			
	100-90%	90-80%	80-70%	70-60%
Based on Goals	5	2	1	2
Based on Schema Details	3	3	2	2

Table 5.14: Resulting grades based on two grading schemes.

In addition, participant problem behavior graphs and demographic sheets were compared to the grades earned using the two schemes. Table 5.15 identifies the participant grades, details for receiving the grades, current computer science courses, grade point averages (GPA), and genders. As shown in the details column in Table 5.15, the participants received either the same or a lower grade based on the quality and features of their schemata. The students received a lower grade for using a while loop instead of a for loop, which is a more efficient choice for the solution to this problem.



Participant	Grade & Demographic Details						
	Based on Goals	Details for Grade	Based on Schema Details	Details for Grade	CPSC	GPA	Gender
2	78%	stdlib.h included -2, no sentinel val -5, didn't init sum var -5, couldn't use sqrt funct. print sqrt -10	78%	same	102	3.0 - 2.5	Male
3	98%	stdlib.h included -2	88%	while loops -10	102	< 2.0	Male
4	63%	return 1 -2, no sentinel val -5, no loops -30	63%	same	212	> 3.5	Male
5	95%	declares new index var for each loop -5, (needed help w/ -lm, used NULL as sentinel)	95%	same	102	3.5 - 3.0	Male
6	93%	stdlib.h included -2, no sentinel val -5, (used calc to check answer)	83%	while loops -10	102	3.0 - 2.5	Male
7	63%	stdlib.h included -2, no return value -5, incorrect Goal 5 & 6 -30, (used NULL as sentinel)	63%	same	212	> 3.5	Male
8	93%	stdlib.h included -2, no sentinel val -5	93%	same	212	3.5 - 3.0	Male
9	93%	stdlib.h included -2, no sentinel val -5	93%	same	212	> 3.5	Male
10	82%	incorrect sentinel value & use, i.e. 15 as sentinel val -3, incorrect loops through arrays -15, (checks answer on paper)	72%	while loops -10	212	> 3.5	Female
11	85%	no sentinel val -5, no loop to calculate sum -10, (needed help w/ -lm)	85%	same	212	3.5 - 3.0	Male

Table 5.15: Results based on two grading scheme, details for receiving the grade, and demographic information per participant.

There were two students who shifted from an A to a B for this reason, and both were enrolled in CPSC 102. However, the two students receiving the lowest grades were CPSC 212 students, and the two receiving the highest grades were in CPSC 102, accounting for the sentinel value ending the actual values. There was only one female participant, a CPSC 212 student who incorrectly used while loops to step through her arrays. This female participant and two other participants with the highest GPAs performed the worst on the problem in this study. Only one of the four participants with the highest GPA scored above a 90%. Those participants with the next highest GPA performed the best, all scoring about 80%, and the student with the lowest GPA performed the best on the problem except for using while loops. At this time, Table 5.15 explains the similarities and differences among participants based on their demographics information and the correctness and quality of schemata. The grades ignore cognitive behavior and the interactions between schemata and metacognition, which might explain the only correlation between grades and the number of interactions between the metacognitive process *start/revisit goals* and the programming strategy *write code*. The participant with the most interactions between mental processes performed the best on the problem, and the two with the least amount of interactions performed the worst. Future research on experts versus novices and additional correlations between behavior and performance can determine new grading schemes, as well as provide support for the predictability of the model.

Chapter 6

Conclusions

The resulting taxonomy and observations from the 10 participants in this research provide initial support for Figure 5.27. This model is in its initial stages of development, and results from the model in research with additional participants on different problems will affect the development of this model. However, the results from the data in this experiment support the hypothesis that solving computer science problems requires the existence of and the interaction among metacognition, schemata, and external sources of information, as illustrated in Figure 5.27. The interactions reported here occur while retrieving and manipulating information when solving a computer science problem. At this time, the model serves as a foundation for explaining how people think in computer science when solving a programming problem, and the executive processes of metacognition appear to provide the dynamics of domain-specific schemata through interactions. All combinations of interactions between metacognition, schemata, and external sources are possible, and generally speaking, people use some type of check/detect strategy before diagnosing an error and diagnose an error before fixing it. In addition, every participant used *understand problem/plan* more than once, and most participants used this process to begin working on the solution. This might be explained by the number of subgoals and goals exceeding seven plus or minus two, which is also known as cognitive load theory (CLT) [116]. The problem behavior graphs provide the details of participant movement through the goals and subgoals in addition to the misconceptions and other errors that can be used to develop multiple-choice tests of reasoning, which are used in physics and chemistry [90] [114]. The errors made by the participants in this research reveal some misconceptions such as not being able to use a for loop with a condition that is array[i] not equal to NULL and being able to initialize array elements by using the array name by itself, i.e. array={1, 2, 3, 4};. In this research, the data supports the hypothesis that metacognition, schemata, and external sources interact when retrieving and manipulating information while solving a computer programming problem by establishing a vocabulary for describing these processes as well as a model for illustrating their interactions.

However, some interactions were never witnessed. The interactions not supported by the data in this research are from *write code* to *diagnose* and from *compile/execute* to *fix code*. Many participants in this research did not use an external source when they verbalized low knowledge, and some participants went directly to writing code or fixing code after their verbalization. CLT might explain why half the students did not address the sentinel value in this problem, but even when some students reread the problem and used an external source to determine the meaning, they still ignored the sentinel value and redefined the problem to meet their needs. These data do not support a correlation between metacognition and schemata and participant performance on this problem, but this may be due to the context by which the metacognitive process is used. Version 6 of the vocabulary for metacognitive and schematic processes in Appendix N attempted to capture finer details and sublevels of these processes, which might be the reason why this research is not showing any correlation to performance.

6.1 Broader Impact

The broader impact of this research can improve learning and pedagogy through a better understanding of the operations on schemata and the validation of additional metacognitive processes that are important for computer science problem solving. In addition, this research provides a model for educators to understand individual and groups of students' mental processes when solving problems in computer science, which has the potential to lower frustration between the educator and student and increase retention rates. An explanation of the differences in how people store, retrieve, and manipulate knowledge while solving and learning computer science problems can help educators facilitate knowledge more effectively and efficiently through a pedagogical focus directed toward student needs rather than toward the educator's perceived student needs. This is accomplished by a focus on the dynamics of the mental processes used to arrive at a solution rather than the analysis of a final solution, and understanding the differences in the dynamics between experts and novices can help expedite the creation of expertise in computer science.

6.2 Future Research

At this time, the research presented here does not address experts versus novices as well as learning, which is responsible for storing knowledge, but the model has implications in the learning sciences. Misconceptions and other errors within schemata and their interactions with metacognition are proposed in the model, but they are not directly related to the hypothesis and can be addressed in future research studies. In addition, this is a sample size of ten and only one problem, but future studies can address different problems and tactics for gathering these mental processes to increase sample size as well as developing a narrower vocabulary for these processes. Provided below is a list of suggested future studies based on the conclusions and potential impact of this research.

- Using the current data, determine if the time spent within each process has a correlation to grade and demographic information.
- Using the current data, determine all misconceptions and other errors in the specific schemata used to solve this problem.
- Using the current data, develop a narrower vocabulary with sublevels and more terms to capture the context of these processes.
- Investigate the interactions within schemata and determine how conceptual and procedural knowledge interact.
- Determine a list of schemata and features used in computer science.
- Collect more data using experts, senior computer students, industry workers, and many problems.
- Collect data from various conditions on memory being accessed, e.g. under time pressure, coffee enhanced, sleep deprived, etc.
- Conduct studies using eye-tracking and other methods in conjuction with verbal protocols to further study peoples thoughts as they solve computer science problems.
- Research computer science problem solving in relationship to Figure 3.1.

- Develop a multiple-choice assessment tool for computer science to capture the collection of processes involved in solving a problem as well as the student's reasoning for using each process.
- Develop refutation texts for computer science providing students with correct solutions and common errors.

6.3 Research Questions

Beyond the immediate future studies resulting from this research, additional pending research questions are listed below.

- What are all the processes in problem solving starting from the problem statement in Figure 3.1?
- How does an individual proceed through the steps involved in problem solving such as parallel steps as the problem statement is read?
- How do the steps in Figure 3.1 relate to other steps of problem solving as defined by Poicare (1913), Wallas (1926), and Polya (1956) as well as others that are a part of the Gestalt theory of problem solving?
- How do experts and novices differ as they proceed through the steps defined in Figure 3.1?
- How does an individual use deductive and inductive reasoning when problem solving?
- How does the static model of a single process of many processes evolve over time?
- How does the static model of a single process relate to and interact with the many parallel processes occurring over time?
- When during the problem-solving process is the problem statement's context determined?
- How is the problem statement's context determined and why?
- How does context determination interact with the model in Figure 5.27?
- When during the problem-solving process is the problem statement chunked?
- How do experts versus novice chunck problem statements?

- How are the chunks organized and processed among individuals?
- How does the chunking process interact with the model in Figure 5.27?
- How does the model in Figure 5.27 interact with each step in Figure 3.1, i.e. schema is used for chunking?
- How exactly is the model in Figure 5.27 activated through the five senses when solving a problem?
- Do experts use the model in Figure 5.27 for solving CS problems, and how does it differ from novices?
- How does instinctual Implicit Knowledge differ from the schemata used for procedural memory as a part of Implicit Knowledge?
- How does problem solving change when external information is removed from the model in Figure 5.27?
- How do experts and novices use Marshall's four properties of schemata differently, and is there a Criteria feature?
- What are the processes associated with learning and schema?
- How are experiences incorporated and used with schema as a part of procedural knowledge?

Appendices

Appendix A Demographics sheet #1

Verbal Protocol – Subject #_____

Demographics

CP SC course: ________ **Gender**: Male Female **Age**:______

School Status: Freshman Sophomore Junior Senior Other

CS/Programming Experience: <1 year 1-2 years 3-5 years >5 years

Overall GPA: ________ **Grade in last CP SC course**: _________

Programming Experience Survey:

What are the different programming languages you know?

How many programming courses have you had?

How many years have you been writing computer programs?

Appendix B Demographics sheet #2

Verbal Protocol – Subject #_____

<u>**Demographics (mark the answer that best applies)**</u>

Current CP SC course: 102 212 **Past CP SC courses**: 101 102 104 111 210

Gender: Male Female **Age**:______ **Major:_________**

School Status: Freshman Sophomore Junior Senior Other________

Ethnicity: Caucasian Asian African-American Latin-American Other______

Reading/Writing English Experience: <5 year 5 - 10 years >10 years

CS/Programming Experience: <1 year 1-2 years 3-5 years >5 years

Overall GPA: >3.5 3.5 - 3.0 3.0 – 2.5 2.5 - 2.0 <2.0

Grade in last CPSC course: A B C Other_______

Are you a first generation college student, i.e. neither parent attended college? Yes No

How many programming courses did you take <u>before</u> attending Clemson University?
1 2 3 4 Other_______

How many programming courses have you <u>completed</u> at Clemson University?
1 2 3 4 Other_______

How many programming languages do you know? 1 2 3 4 Other_______
List:____________________________________

How many years of professional work experience do you have?
0 1 2 3 4 5 6 7 8 9 10 Other

Appendix C IRB Informational Letter

Information Concerning Participation in a Research Study Clemson University

Test of Computational Thinking

Description of the research and your participation

You are invited to participate in a research study conducted by a computer science professor, Dr. D. E. Stevenson, and PhD candidate, Jennifer R. Parham. The purpose of this research is to collect cognitive data used to solve computing problems from students enrolled in a computer science course. This study is used for research purposes only. The research involves testing computational thinking skills while taking a computer science course. The results will be used to assess the students schematic and metacognitive knowledge and evaluate the correlation between schemata and metacognition in computational thinking. There will be about 20-25 participants per year for the next year of this study.

Your participation will involve solving one to three computing problems or case studies. Each case study or problem may ask a series of questions and ask you to verbalize your thoughts as you solve the problem by thinking aloud. The voice recording of your thoughts will be analyzed to measure the correlation between schematic and metacognitive knowledge in the computing sciences.

The amount of time required for your participation will vary depending on the number of problems or case studies presented, but your participation will not require more than 60 minutes.

Risks and discomforts

There are no known risks associated with this research. The data will not have any identifying information, such as Clemson IDs, Social Security #s, names, or addresses, attached to the test results.

Potential benefits

Students may receive a better computer science education resulting from this research. The results will explain the potential errors in student computing knowledge used to solve problems and ultimately affect learning through new testing strategies and teaching methods. This research may help us to understand the interactions between schemata and metacognition through specific processes involved in solving computer science problems.

Protection of confidentiality

We will do everything we can to protect your privacy in this research. We will not ask for any identifiable information other than general demographic information, such as gender, race, and class status. Your identity will not be revealed in any publication that might result from this study.

Voluntary participation

Your participation in this research study is voluntary. You may choose not to participate and you may withdraw your consent to participate at any time. You will not be penalized in any way should you decide not to participate or to withdraw from this study.

Contact information

If you have any questions or concerns about this study or if any problems arise, please contact Dr. D. E. Stevenson at Clemson University at 864-656-5880. If you have any questions or concerns about your rights as a research participant, please contact the Clemson University Office of Research Compliance at 864.656.6460.

Appendix D Problem and Goals/Schemata - Version 1

Problem Statement:

Write a program to take two numerical lists of the same length ended by a sentinel value and store the lists in arrays x and y, each of which can hold a maximum number of 10 elements, and populate both lists by entering the actual data values from within the program. Let n represent the actual number of data values in each list. For example, x and y can hold up to 10 elements, but the two arrays may only contain 5 actual data values, i.e. n=5. Store the product of corresponding elements of x and y in a third array z, also of maximum size 10. Print the contents of the arrays x, y, and z. Then compute and print the square root of the sum of the items in z.

Goal 1 Begin program

Subtasks:

1. Create/open program file
2. write main routine

Goal 2 Create arrays

Subtasks:

1. create x, y, and z with 10 elements
2. populate/initialize the x and y array
3. add a sentinel value at the end.

Goal 3 Store product of x and y in z.

Subtasks:

1. create loop to go until sentinel value or n found by sentinel value
2. multiply x[i] and y[i] and assign to z[i]

Goal 4 Print arrays

Subtasks:

1. include stdio.h (if not already included)
2. create loop to go until sentinel value or n found by sentinel value**
3. print x[i], y[i], and z[i].

*Can use three loops or one loop to print x, y, and z arrays.**

Goal 5 Compute square root of sum of z.

Subtasks:

1. include math.h (if not already included)
2. create a sum variable
3. initialize sum variable to 0
4. create loop to go until sentinel value or n found by sentinel value
5. sum variable gets z[i] plus previous sum value
6. find square root of sum.

Goal 6 Print square root of sum of z.

Subtasks:

1. print square root of sum.

Appendix E Problem and Goals/Schemata - Version 2

Problem Statement:

Write a program to take two numerical lists of the same length ended by a sentinel value and store the lists in arrays x and y, each of which can hold a maximum number of 10 elements, and populate both lists by entering the actual data values from within the program. Let n represent the actual number of data values in each list. For example, x and y can hold up to 10 elements, but the two arrays may only contain 5 actual data values, i.e. n=5. Store the product of corresponding elements of x and y in a third array z, also of maximum size 10. Print the contents of the arrays x, y, and z. Then compute and print the square root of the sum of the items in z.

Goals:

- **Subtasks in bullets**

Goal 1 Begin Program

- Construct main routine
- Include library files

Goal 2 Create/Initialize Data Structures

- Declare x, y, and z with 10 elements
- Populate the x and y array w/ data values ended by sentinel value
- Declare other variables
- Initialize other variables

Goal 3 Compute/Store product of x and y in z.

- Construct loop to step through arrays
- Multiply elements in x and y and assign to z

Goal 4 Print arrays

- Construct loop to step through arrays

- Construct print statement to output values

Goal 5 Compute square root of sum of z.

- Construct loop to step through arrays

- Calculate the sum of the values in the z array

- Use square root function

Goal 6 Print square root of sum of z.

- Construct print statement to output values

Appendix F Subject 6 Morae Produced .CSV Data (First 10 minutes)

00:00.0				problem	[E informed P that we were recording and he could ask any questions] Where do I click to ge
00:08.2	Mouse Clic	putty.exe	Windows E	Running Applications	
00:10.6	Keystrokes		login		
00:20.0	Keystrokes			ls	
00:25.4	Keystrokes			clear	
00:32.8	Keystrokes			Ctrl c	
00:37.2	Keystrokes			clear	
00:41.8	Keystrokes			mkdir test	
00:48.6	Keystrokes			cd test	
00:52.4	Keystrokes			nano test.c	
01:16.0	Keystrokes			#include "math.h	
01:23.8	Mouse Clic	Windows E	L Button D	FolderView	
01:24.2	Keystrokes	Windows E		Left Arrow	
01:25.2	Mouse Clic	putty.exe	Windows E	Running Applications	
01:26.6	Keystrokes			Left Arrow	
01:26.8	Keystrokes			Left Arrow	
01:26.8	Keystrokes			Left Arrow	
01:27.0	Keystrokes			Left Arrow	
01:27.2	Keystrokes			Left Arrow	
01:27.4	Keystrokes			Left Arrow	
01:28.0	Keystrokes			Backspace	
01:28.4	Keystrokes		Shift	Shift	
01:28.4	Keystrokes		Shift	<	
01:29.0	Keystrokes			Right Arrow	
01:29.0	Keystrokes			Right Arrow	
01:29.2	Keystrokes			Right Arrow	
01:29.4	Keystrokes			Right Arrow	
01:29.6	Keystrokes			Right Arrow	
01:29.6	Keystrokes			Right Arrow	
01:30.2	Keystrokes		Shift	Shift	
01:30.4	Keystrokes		Shift	>	
01:30.6	Keystrokes			Enter	
01:31.6	Keystrokes		Shift	Shift	
01:32.0	Keystrokes		Shift	Shift	
01:32.2	Keystrokes		Shift	Shift	
01:32.2	Keystrokes		Shift	Shift	
01:32.2	Keystrokes		Shift	Shift	
01:32.2	Keystrokes		Shift	Shift	
01:32.2	Keystrokes		Shift	Shift	
01:32.2	Keystrokes		Shift	#	
01:32.6	Keystrokes			i	
01:33.4	Keystrokes			c	
01:34.2	Keystrokes			Backspace	
01:34.4	Keystrokes			n	
01:34.8	Keystrokes			c	
01:35.2	Keystrokes			l	
01:35.4	Keystrokes			u	
01:35.6	Keystrokes			d	
01:36.0	Keystrokes			e	
01:36.0	Keystrokes			Spacebar	
01:36.6	Keystrokes		Shift	Shift	
01:36.8	Keystrokes		Shift	<	
01:37.0	Keystrokes			s	
01:37.4	Keystrokes			t	
01:37.8	Keystrokes			d	
01:37.8	Keystrokes			i	
01:38.0	Keystrokes			o	
01:38.6	Keystrokes			.	
01:40.2	Keystrokes			h	
01:40.6	Keystrokes		Shift	Shift	
01:40.6	Keystrokes		Shift	>	
01:41.2	Keystrokes			Enter	
01:44.8	Keystrokes		Shift	Shift	
01:45.2	Keystrokes		Shift	#	
01:45.6	Keystrokes			Backspace	
01:48.8	Keystrokes			Up Arrow	
01:49.0	Keystrokes			Right Arrow	
01:49.6	Keystrokes			Right Arrow	
01:49.6	Keystrokes			Right Arrow	
01:49.6	Keystrokes			Right Arrow	
01:49.6	Keystrokes			Right Arrow	
01:49.6	Keystrokes			Right Arrow	
01:49.8	Keystrokes			Right Arrow	
01:49.8	Keystrokes			Right Arrow	

```
01:49.8 Keystrokes                      Right Arrow
01:49.8 Keystrokes                      Right Arrow
01:49.8 Keystrokes                      Right Arrow
01:49.8 Keystrokes                      Right Arrow
01:50.0 Keystrokes                      Right Arrow
01:50.0 Keystrokes                      Right Arrow
01:50.0 Keystrokes                      Right Arrow
01:50.0 Keystrokes                      Right Arrow
01:50.0 Keystrokes                      Right Arrow
01:50.0 Keystrokes                      Right Arrow
01:50.2 Keystrokes                      Left Arrow
01:50.4 Keystrokes                      Left Arrow
01:50.6 Keystrokes                      Left Arrow
01:51.2 Keystrokes                      Backspace
01:51.4 Keystrokes                      Backspace
01:52.2 Keystrokes                      l
01:52.4 Keystrokes                      i
01:52.8 Keystrokes                      d
01:53.2 Keystrokes                      Backspace
01:53.4 Keystrokes                      Backspace
01:54.6 Keystrokes                      i
01:54.8 Keystrokes                      b
01:55.8 Keystrokes                      Right Arrow
01:56.0 Keystrokes                      Right Arrow
01:56.0 Keystrokes                      Right Arrow
01:56.6 Keystrokes                      Enter
01:56.8 Keystrokes                      Enter
01:58.4 Keystrokes                      i
01:58.4 Keystrokes                      n
01:58.6 Keystrokes                      t
01:58.8 Keystrokes                      Spacebar
02:03.6 Keystrokes                      m
02:03.8 Keystrokes                      a
02:03.8 Keystrokes                      i
02:03.8 Keystrokes                      n
02:04.4 Keystrokes                      Spacebar
02:05.0 Keystrokes                      Backspace
02:05.2 Keystrokes            Shift     Shift
02:05.4 Keystrokes            Shift     (
02:05.6 Keystrokes            Shift     )
02:06.2 Keystrokes            Shift     Shift
02:06.4 Keystrokes            Shift     {
02:06.8 Keystrokes                      Enter
02:07.2 Keystrokes                      Enter
02:18.4 Keystrokes                      i
02:18.4 Keystrokes                      n
02:18.6 Keystrokes                      t
02:18.6 Keystrokes                      Spacebar
02:19.2 Keystrokes                      Backspace
02:41.4 Keystrokes                      Spacebar
02:41.6 Keystrokes                      x
02:42.6 Keystrokes                      [
02:44.2 Keystrokes                                 1
02:45.4 Keystrokes                                 0
02:46.4 Keystrokes                      ]
02:50.4 Keystrokes                      Spacebar
02:50.6 Keystrokes                      =
02:50.8 Keystrokes                      Spacebar
02:51.6 Keystrokes            Shift     Shift
02:51.8 Keystrokes            Shift     {
02:53.6 Keystrokes                                 1
02:53.6 Keystrokes                      ,
02:53.8 Keystrokes                      Spacebar
02:54.6 Keystrokes                                 2
02:55.0 Keystrokes                      ,
02:55.2 Keystrokes                      Spacebar
02:55.8 Keystrokes                                 3
02:58.4 Keystrokes                      ,
02:58.6 Keystrokes                      Spacebar
02:58.8 Keystrokes                                 4
03:00.0 Keystrokes                      ,
03:00.2 Keystrokes                      Spacebar
03:00.4 Keystrokes                                 5
03:01.2 Keystrokes            Shift     Shift
03:01.4 Keystrokes          Shift     }
03:02.4 Keystrokes                    ;
03:02.6 Keystrokes                    Enter
03:03.8 Keystrokes                    i
03:04.0 Keystrokes                    n
03:04.0 Keystrokes                    t
03:04.2 Keystrokes                    Spacebar
03:05.0 Keystrokes                    y
03:05.2 Keystrokes                    Spacebar
03:05.6 Keystrokes                    Backspace
03:06.4 Keystrokes                    [
03:07.4 Keystrokes                           1
03:07.6 Keystrokes                           0
03:08.8 Keystrokes                    ]
03:09.4 Keystrokes                    Spacebar
03:10.0 Keystrokes                    =
03:10.2 Keystrokes                    Spacebar
03:10.8 Keystrokes          Shift     Shift
03:11.0 Keystrokes          Shift     {
03:11.4 Keystrokes                           1
03:11.6 Keystrokes                    ,
03:11.8 Keystrokes                    Spacebar
03:12.0 Keystrokes                           2
03:12.4 Keystrokes                    ,
03:12.4 Keystrokes                    Spacebar
03:14.0 Keystrokes                    Backspace
03:14.2 Keystrokes                    Backspace
03:14.4 Keystrokes                    Backspace
03:14.6 Keystrokes                    Backspace
03:14.6 Keystrokes                    Backspace
03:14.8 Keystrokes                    Backspace
03:16.0 Keystrokes                           4
03:16.2 Keystrokes                    ,
03:16.4 Keystrokes                    Spacebar
03:17.2 Keystrokes                           5
03:17.4 Keystrokes                    ,
03:17.4 Keystrokes                    Spacebar
03:17.6 Keystrokes                    ,
03:17.8 Keystrokes                           6
03:18.0 Keystrokes                    Spacebar
03:18.6 Keystrokes                    Backspace
03:18.8 Keystrokes                    Backspace
03:19.0 Keystrokes                    Backspace
03:19.6 Keystrokes                           6
03:20.4 Keystrokes                    ,
03:20.6 Keystrokes                    Spacebar
03:21.2 Keystrokes                           7
03:21.8 Keystrokes                    ,
03:22.4 Keystrokes                    Spacebar
03:23.4 Keystrokes                           8
03:23.8 Keystrokes          Shift     Shift
03:24.0 Keystrokes          Shift     }
03:25.2 Keystrokes                    ;
03:25.4 Keystrokes                    Enter
03:25.8 Keystrokes                    Enter
03:38.0 Keystrokes                    i
03:38.2 Keystrokes                    n
03:38.4 Keystrokes                    t
03:38.8 Keystrokes                    Spacebar
03:39.4 Keystrokes                    Backspace
03:39.8 Keystrokes                    Spacebar
03:40.4 Keystrokes                    z
03:40.6 Keystrokes                    Spacebar
03:41.6 Keystrokes                    Backspace
03:42.0 Keystrokes                    [
03:42.8 Keystrokes                           1
03:43.4 Keystrokes                           0
03:43.8 Keystrokes                    ]
03:45.2 Keystrokes                    ;
04:08.0 Keystrokes                    Enter
04:08.2 Keystrokes                    Enter
04:09.0 Keystrokes                    i
04:09.2 Keystrokes                    n
04:09.2 Keystrokes                    t
```

04:09.4	Keystrokes		Spacebar
04:09.6	Keystrokes		n
04:10.6	Keystrokes		;
04:11.8	Keystrokes		Enter
04:12.0	Keystrokes		Enter
04:21.2	Keystrokes		Backspace
04:21.6	Keystrokes		Backspace
04:21.8	Keystrokes		Backspace
04:25.8	Keystrokes		Spacebar
04:26.2	Keystrokes		=
04:26.4	Keystrokes		Spacebar
04:27.4	Keystrokes		5
04:28.8	Keystrokes		;
04:29.2	Keystrokes		Enter
04:29.4	Keystrokes		Enter
04:36.8	Keystrokes		f
04:37.0	Keystrokes		o
04:37.8	Keystrokes		r
04:40.6	Mouse Clicks	L Button D	Software Update
04:45.6	Keystrokes		Backspace
04:45.8	Keystrokes		Backspace
04:46.0	Keystrokes		Backspace
04:46.4	Keystrokes		Backspace
04:47.4	Keystrokes		i
04:47.4	Keystrokes		n
04:48.2	Keystrokes		t
04:48.4	Keystrokes		Spacebar
04:48.6	Keystrokes		i
04:50.0	Keystrokes		;
04:50.4	Keystrokes		Enter
04:50.6	Keystrokes		Enter
04:51.4	Keystrokes		w
04:51.6	Keystrokes		h
04:51.6	Keystrokes		i
04:51.8	Keystrokes		l
04:52.2	Keystrokes		e
04:52.2	Keystrokes		Spacebar
04:53.0	Keystrokes	Shift	Shift
04:53.2	Keystrokes	Shift	(
04:53.8	Keystrokes		i
04:54.6	Keystrokes		Spacebar
04:54.8	Keystrokes	Shift	Shift
04:55.0	Keystrokes	Shift	<
04:55.6	Keystrokes		Spacebar
04:58.2	Keystrokes		5
04:58.4	Mouse Clicks	L Button D	spider5.cs.clemson.edu - PuTTY
05:00.6	Keystrokes		Left Arrow
05:00.8	Keystrokes		Right Arrow
05:01.4	Keystrokes		Backspace
05:01.6	Keystrokes		4
05:02.4	Keystrokes	Shift	Shift
05:02.4	Keystrokes	Shift	)
05:03.0	Keystrokes		Up Arrow
05:03.2	Keystrokes		Left Arrow
05:03.4	Keystrokes		Up Arrow
05:03.8	Keystrokes		Down Arrow
05:04.0	Keystrokes		Left Arrow
05:04.8	Keystrokes		Spacebar
05:05.0	Keystrokes		=
05:05.4	Keystrokes		Spacebar
05:06.2	Keystrokes		0
05:07.0	Keystrokes		Down Arrow
05:07.2	Keystrokes		Down Arrow
05:07.4	Keystrokes		Down Arrow
05:08.0	Keystrokes		Up Arrow
05:09.0	Keystrokes		Down Arrow
05:09.4	Keystrokes		Right Arrow
05:11.0	Keystrokes		Backspace
05:11.4	Keystrokes		Backspace
05:12.0	Keystrokes	Shift	Shift
05:12.2	Keystrokes	Shift	{
05:12.6	Keystrokes		Enter
05:12.8	Keystrokes		Enter
05:21.2	Keystrokes		x

05:23.6	Keystrokes	Shift	Shift
05:24.0	Keystrokes	Shift	Shift
05:24.2	Keystrokes	Shift	Shift
05:24.2	Keystrokes	Shift	Shift
05:24.2	Keystrokes	Shift	Shift
05:24.2	Keystrokes	Shift	Shift
05:24.2	Keystrokes	Shift	Shift
05:24.4	Keystrokes	Shift	Shift
05:24.4	Keystrokes	Shift	Shift
05:24.4	Keystrokes	Shift	Shift
05:24.4	Keystrokes	Shift	Shift
05:24.4	Keystrokes	Shift	Shift
05:24.4	Keystrokes	Shift	Shift
05:24.6	Keystrokes	Shift	Shift
05:24.6	Keystrokes	Shift	Shift
05:24.6	Keystrokes	Shift	Shift
05:24.6	Keystrokes	Shift	Shift
05:24.6	Keystrokes	Shift	Shift
05:24.6	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:24.8	Keystrokes	Shift	Shift
05:25.0	Keystrokes	Shift	Shift
05:25.0	Keystrokes	Shift	Shift
05:25.0	Keystrokes	Shift	Shift
05:25.0	Keystrokes	Shift	Shift
05:25.0	Keystrokes	Shift	Shift
05:25.0	Keystrokes	Shift	Shift
05:25.2	Keystrokes	Shift	Shift
05:25.2	Keystrokes	Shift	Shift
05:25.2	Keystrokes	Shift	Shift
05:25.2	Keystrokes	Shift	Shift
05:25.2	Keystrokes	Shift	Shift
05:25.4	Keystrokes	Shift	Shift
05:25.4	Keystrokes	Shift	{
05:26.0	Keystrokes		Backspace
05:26.8	Keystrokes		[
05:27.0	Keystrokes		i
05:28.0	Keystrokes		Backspace
05:28.0	Keystrokes		Backspace
05:28.2	Keystrokes		Backspace
05:29.0	Keystrokes		z
05:30.4	Keystrokes		[
05:31.6	Keystrokes		i
05:32.8	Keystrokes		]
05:33.0	Keystrokes		Spacebar
05:33.4	Keystrokes		=
05:33.6	Keystrokes		Spacebar
05:34.4	Keystrokes		x
05:35.8	Keystrokes		[
05:37.6	Keystrokes		i
05:37.8	Keystrokes		Spacebar
05:38.2	Keystrokes		Backspace
05:38.8	Keystrokes		]
05:39.0	Keystrokes		Spacebar
05:39.4	Keystrokes	Shift	Shift
05:40.0	Keystrokes	Shift	*
05:40.0	Keystrokes		Spacebar
05:41.8	Keystrokes		y
05:42.0	Keystrokes		Spacebar
05:42.8	Keystrokes		Backspace
05:43.0	Keystrokes		[
05:44.2	Keystrokes		i
05:45.4	Keystrokes		]
05:46.4	Keystrokes		;
05:46.6	Keystrokes		Enter
05:47.6	Keystrokes		i
05:48.2	Keystrokes	Shift	Shift
05:48.4	Keystrokes	Shift	+
05:48.6	Keystrokes	Shift	+

05:49.4	Keystrokes		;
05:51.0	Keystrokes		Enter
05:51.8	Keystrokes	Shift	Shift
05:52.0	Keystrokes	Shift	}
05:52.4	Keystrokes		Enter
06:11.8	Keystrokes		Down Arrow
06:13.8	Keystrokes		w
06:14.0	Keystrokes		h
06:14.2	Keystrokes		i
06:14.2	Keystrokes		l
06:14.4	Keystrokes		e
06:14.4	Keystrokes		Spacebar
06:14.8	Keystrokes		Backspace
06:15.2	Keystrokes		Spacebar
06:15.4	Keystrokes	Shift	Shift
06:15.8	Keystrokes	Shift	(
06:18.2	Keystrokes		Backspace
06:18.4	Keystrokes		Backspace
06:19.0	Keystrokes		Backspace
06:19.0	Keystrokes		Backspace
06:19.0	Keystrokes		Backspace
06:19.0	Keystrokes		Backspace
06:19.0	Keystrokes		Backspace
06:19.0	Keystrokes		Backspace
06:19.2	Keystrokes		Backspace
06:19.2	Keystrokes		Backspace
06:19.2	Keystrokes		Backspace
06:20.6	Keystrokes		Enter
06:20.8	Keystrokes		Enter
06:21.4	Keystrokes	Shift	Shift
06:21.4	Keystrokes	Shift	}
06:21.6	Keystrokes		Enter
06:22.4	Keystrokes		Enter
06:23.0	Keystrokes		i
06:23.0	Keystrokes		Spacebar
06:23.6	Keystrokes		=
06:23.6	Keystrokes		Spacebar
06:23.8	Keystrokes		0
06:25.0	Keystrokes		;
06:25.4	Keystrokes		Enter
06:25.8	Keystrokes		Enter
06:26.4	Keystrokes		w
06:26.4	Keystrokes		h
06:26.6	Keystrokes		i
06:26.6	Keystrokes		l
06:26.8	Keystrokes		e
06:27.6	Keystrokes		Spacebar
06:28.4	Keystrokes	Shift	Shift
06:28.6	Keystrokes	Shift	(
06:29.4	Keystrokes		i
06:29.6	Keystrokes	Shift	Shift
06:30.2	Keystrokes	Shift	Shift
06:30.2	Keystrokes	Shift	Shift
06:30.2	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.4	Keystrokes	Shift	Shift
06:30.6	Keystrokes	Shift	Shift
06:30.6	Keystrokes	Shift	Shift
06:30.6	Keystrokes	Shift	<
06:31.2	Keystrokes		4
06:33.0	Keystrokes	Shift	Shift
06:33.2	Keystrokes	Shift	)
06:34.2	Keystrokes	Shift	Shift
06:34.4	Keystrokes	Shift	{
06:34.6	Keystrokes		Enter
06:35.0	Keystrokes		Enter
06:38.4	Keystrokes		f
06:38.4	Keystrokes		p
06:38.6	Keystrokes		r

```
06:38.8 Keystrokes                  i
06:38.8 Keystrokes                  n
06:39.0 Keystrokes                  t
06:39.6 Keystrokes                  f
06:40.0 Keystrokes          Shift   Shift
06:40.4 Keystrokes          Shift   (
06:41.0 Keystrokes                  s
06:41.2 Keystrokes                  t
06:41.6 Keystrokes                  d
06:41.6 Keystrokes                  o
06:42.0 Keystrokes                  u
06:42.2 Keystrokes                  t
06:42.4 Keystrokes                  ,
06:42.6 Keystrokes                  Spacebar
06:43.6 Keystrokes          Shift   Shift
06:43.8 Keystrokes          Shift   '
06:46.2 Keystrokes          Shift   Shift
06:46.6 Keystrokes          Shift   %
06:47.2 Keystrokes                  d
06:47.4 Keystrokes                  Spacebar
06:48.8 Keystrokes                  Backspace
06:49.0 Keystrokes                  Backspace
06:49.2 Keystrokes                  Backspace
07:00.8 Keystrokes                  Spacebar
07:01.6 Keystrokes          Shift   Shift
07:01.6 Keystrokes          Shift   A
07:02.0 Keystrokes                  r
07:02.2 Keystrokes                  r
07:02.4 Keystrokes                  a
07:02.6 Keystrokes                  y
07:03.8 Keystrokes                  Left Arrow
07:03.8 Keystrokes                  Left Arrow
07:04.0 Keystrokes                  Left Arrow
07:04.2 Keystrokes                  Left Arrow
07:04.4 Keystrokes                  Left Arrow
07:04.8 Keystrokes                  Backspace
07:05.4 Keystrokes                  Right Arrow
07:05.6 Keystrokes                  Right Arrow
07:05.8 Keystrokes                  Right Arrow
07:05.8 Keystrokes                  Right Arrow
07:06.2 Keystrokes                  Right Arrow
07:06.6 Keystrokes                  Spacebar
07:08.0 Keystrokes          Shift   Shift
07:08.2 Keystrokes          Shift   X
07:08.6 Keystrokes          Shift   Shift
07:08.6 Keystrokes          Shift   :
07:09.0 Keystrokes                  Spacebar
07:09.8 Keystrokes          Shift   Shift
07:10.0 Keystrokes          Shift   A
07:10.2 Keystrokes                  r
07:10.4 Keystrokes                  r
07:10.8 Keystrokes                  a
07:10.8 Keystrokes                  y
07:11.6 Keystrokes                  Spacebar
07:12.4 Keystrokes                  Backspace
07:12.4 Keystrokes                  Backspace
07:12.6 Keystrokes                  Backspace
07:12.8 Keystrokes                  Backspace
07:12.8 Keystrokes                  Backspace
07:13.0 Keystrokes                  Backspace
07:13.4 Keystrokes                  Spacebar
07:13.6 Keystrokes                  Spacebar
07:14.0 Keystrokes          Shift   Shift
07:14.2 Keystrokes          Shift   A
07:14.4 Keystrokes                  r
07:14.6 Keystrokes                  r
07:14.8 Keystrokes                  a
07:15.0 Keystrokes                  y
07:15.4 Keystrokes                  Spacebar
07:15.8 Keystrokes          Shift   Shift
07:15.8 Keystrokes          Shift   U
07:16.2 Keystrokes                  Backspace
07:16.4 Keystrokes                  y
07:17.0 Keystrokes                  Backspace
```

07:17.2	Keystrokes	Shift	Shift
07:17.2	Keystrokes	Shift	Y
07:17.8	Keystrokes	Shift	Shift
07:18.2	Keystrokes	Shift	:
07:18.6	Keystrokes		Spacebar
07:19.2	Keystrokes	Shift	Shift
07:19.4	Keystrokes	Shift	A
07:19.8	Keystrokes		Backspace
07:20.0	Keystrokes		Spacebar
07:20.2	Keystrokes		Spacebar
07:20.6	Keystrokes	Shift	Shift
07:20.8	Keystrokes	Shift	A
07:21.0	Keystrokes		r
07:21.4	Keystrokes		r
07:21.8	Keystrokes		a
07:22.4	Keystrokes		Backspace
07:22.6	Keystrokes		Backspace
07:22.8	Keystrokes		Backspace
07:22.8	Keystrokes		Backspace
07:23.4	Keystrokes	Shift	Shift
07:23.6	Keystrokes	Shift	P
07:24.0	Keystrokes		r
07:24.2	Keystrokes		o
07:24.6	Keystrokes		Backspace
07:25.4	Keystrokes		o
07:25.6	Keystrokes		d
07:25.6	Keystrokes		u
07:26.0	Keystrokes		c
07:26.4	Keystrokes		t
07:28.6	Keystrokes		Spacebar
07:28.6	Keystrokes		o
07:29.0	Keystrokes		f
07:29.0	Keystrokes		Spacebar
07:29.8	Keystrokes	Shift	Shift
07:30.0	Keystrokes	Shift	X
07:30.6	Keystrokes		Spacebar
07:33.0	Keystrokes		a
07:33.2	Keystrokes		n
07:34.6	Keystrokes		d
07:35.2	Keystrokes		Spacebar
07:35.8	Keystrokes	Shift	Shift
07:36.2	Keystrokes	Shift	Y
07:42.8	Keystrokes	Shift	Shift
07:43.2	Keystrokes	Shift	Shift
07:43.2	Keystrokes	Shift	Shift
07:43.4	Keystrokes	Shift	Shift
07:43.4	Keystrokes	Shift	Shift
07:43.4	Keystrokes	Shift	Shift
07:43.4	Keystrokes	Shift	Shift
07:43.4	Keystrokes	Shift	Shift
07:43.4	Keystrokes	Shift	Shift
07:43.6	Keystrokes	Shift	Shift
07:43.6	Keystrokes	Shift	Shift
07:43.6	Keystrokes	Shift	Shift
07:43.6	Keystrokes	Shift	Shift
07:43.6	Keystrokes	Shift	Shift
07:43.6	Keystrokes	Shift	Shift
07:43.8	Keystrokes	Shift	Shift
07:43.8	Keystrokes	Shift	Shift
07:43.8	Keystrokes	Shift	Shift
07:43.8	Keystrokes	Shift	Shift
07:43.8	Keystrokes	Shift	Shift
07:43.8	Keystrokes	Shift	Shift
07:44.0	Keystrokes	Shift	Shift
07:44.0	Keystrokes	Shift	Shift
07:44.0	Keystrokes	Shift	Shift
07:44.0	Keystrokes	Shift	Shift
07:44.0	Keystrokes	Shift	Shift
07:46.2	Keystrokes		Spacebar
07:48.4	Keystrokes		Spacebar
07:50.2	Keystrokes		Backspace
07:50.8	Keystrokes		Backspace
07:50.8	Keystrokes		Backspace
07:50.8	Keystrokes		Backspace

07:50.8	Keystrokes		Backspace
07:50.8	Keystrokes		Backspace
07:50.8	Keystrokes		Backspace
07:51.0	Keystrokes		Backspace
07:51.0	Keystrokes		Backspace
07:51.0	Keystrokes		Backspace
07:51.0	Keystrokes		Backspace
07:51.0	Keystrokes		Backspace
07:51.0	Keystrokes		Backspace
07:51.2	Keystrokes		Backspace
07:51.2	Keystrokes		Backspace
07:51.2	Keystrokes		Backspace
07:51.2	Keystrokes		Backspace
07:51.2	Keystrokes		Backspace
07:51.6	Keystrokes		Backspace
07:51.8	Keystrokes		Backspace
07:52.0	Keystrokes	Shift	Shift
07:52.2	Keystrokes	Shift	A
07:52.4	Keystrokes		r
07:52.6	Keystrokes		r
07:53.0	Keystrokes		a
07:53.0	Keystrokes		y
07:53.4	Keystrokes		Spacebar
07:54.0	Keystrokes	Shift	Shift
07:54.2	Keystrokes	Shift	X
07:54.2	Keystrokes	Shift	Z
07:55.0	Keystrokes		Backspace
07:55.2	Keystrokes		Backspace
07:55.4	Keystrokes	Shift	Shift
07:55.6	Keystrokes	Shift	Z
07:56.6	Keystrokes		Spacebar
07:56.8	Keystrokes	Shift	Shift
07:57.0	Keystrokes	Shift	(
07:57.4	Keystrokes	Shift	Shift
07:57.6	Keystrokes	Shift	P
07:57.8	Keystrokes		r
07:58.0	Keystrokes		o
07:58.6	Keystrokes		d
07:59.0	Keystrokes		u
07:59.2	Keystrokes		c
07:59.6	Keystrokes		t
07:59.8	Keystrokes		Spacebar
07:59.8	Keystrokes		o
08:00.0	Keystrokes		f
08:00.0	Keystrokes		Spacebar
08:00.4	Keystrokes	Shift	Shift
08:00.6	Keystrokes	Shift	X
08:00.6	Keystrokes		Spacebar
08:01.0	Keystrokes		a
08:01.0	Keystrokes		n
08:01.2	Keystrokes		d
08:01.4	Keystrokes		Spacebar
08:01.6	Keystrokes	Shift	Shift
08:01.6	Keystrokes	Shift	W
08:01.8	Keystrokes		h
08:02.0	Keystrokes		i
08:02.0	Keystrokes		l
08:02.2	Keystrokes		e
08:02.4	Keystrokes		Backspace
08:02.6	Keystrokes		Backspace
08:02.8	Keystrokes		Backspace
08:02.8	Keystrokes		Backspace
08:03.2	Keystrokes		Backspace
08:03.6	Keystrokes	Shift	Shift
08:04.0	Keystrokes	Shift	Y
08:04.6	Keystrokes	Shift	Shift
08:04.8	Keystrokes	Shift	)
08:06.6	Keystrokes		Left Arrow
08:07.2	Keystrokes		Left Arrow
08:07.2	Keystrokes		Left Arrow
08:07.2	Keystrokes		Left Arrow
08:07.4	Keystrokes		Left Arrow
08:07.4	Keystrokes		Left Arrow
08:07.4	Keystrokes		Left Arrow

08:07.4	Keystrokes		Left Arrow
08:07.4	Keystrokes		Left Arrow
08:07.4	Keystrokes		Left Arrow
08:07.4	Keystrokes		Left Arrow
08:07.6	Keystrokes		Left Arrow
08:07.6	Keystrokes		Left Arrow
08:07.6	Keystrokes		Left Arrow
08:07.6	Keystrokes		Left Arrow
08:07.6	Keystrokes		Left Arrow
08:07.6	Keystrokes		Left Arrow
08:07.8	Keystrokes		Left Arrow
08:07.8	Keystrokes		Left Arrow
08:07.8	Keystrokes		Left Arrow
08:07.8	Keystrokes		Left Arrow
08:07.8	Keystrokes		Left Arrow
08:07.8	Keystrokes		Left Arrow
08:08.0	Keystrokes		Left Arrow
08:08.0	Keystrokes		Right Arrow
08:08.2	Keystrokes		Right Arrow
08:08.6	Keystrokes		Right Arrow
08:09.2	Keystrokes	Shift	Shift
08:09.6	Keystrokes	Shift	:
08:09.6	Mouse Clicks	L Button Do	FolderView
08:10.2	Keystrokes		Right Arrow
08:11.6	Mouse Clicks	L Button Do	spider5.cs.clemson.edu - PuTTY
08:13.0	Keystrokes		Right Arrow
08:13.4	Keystrokes		Right Arrow
08:13.4	Keystrokes		Right Arrow
08:13.6	Keystrokes		Right Arrow
08:13.6	Keystrokes		Right Arrow
08:13.6	Keystrokes		Right Arrow
08:13.6	Keystrokes		Right Arrow
08:13.6	Keystrokes		Right Arrow
08:13.6	Keystrokes		Right Arrow
08:13.8	Keystrokes		Right Arrow
08:13.8	Keystrokes		Right Arrow
08:13.8	Keystrokes		Right Arrow
08:13.8	Keystrokes		Right Arrow
08:13.8	Keystrokes		Right Arrow
08:13.8	Keystrokes		Right Arrow
08:14.2	Keystrokes		Right Arrow
08:14.4	Keystrokes		Right Arrow
08:14.4	Keystrokes		Right Arrow
08:14.6	Keystrokes		Right Arrow
08:14.8	Keystrokes		Right Arrow
08:15.0	Keystrokes		Right Arrow
08:15.8	Keystrokes		Spacebar
08:18.2	Keystrokes	Shift	Shift
08:18.6	Keystrokes	Shift	S
08:18.8	Keystrokes		q
08:19.2	Keystrokes		u
08:19.4	Keystrokes		a
08:19.8	Keystrokes		r
08:20.0	Keystrokes		e
08:20.2	Keystrokes		Spacebar
08:20.6	Keystrokes		r
08:20.6	Keystrokes		o
08:20.8	Keystrokes		o
08:21.0	Keystrokes		t
08:21.0	Keystrokes		Spacebar
08:21.2	Keystrokes		o
08:21.4	Keystrokes		f
08:21.4	Keystrokes		Spacebar
08:21.8	Keystrokes	Shift	Shift
08:22.0	Keystrokes	Shift	Z
08:23.0	Keystrokes	Shift	Shift
08:23.2	Keystrokes	Shift	:
08:23.6	Keystrokes	Shift	Shift
08:23.6	Keystrokes	Shift	'
08:25.4	Keystrokes	Shift	Shift
08:25.8	Keystrokes	Shift	)
08:27.0	Keystrokes		;
08:27.8	Keystrokes		Enter
08:28.2	Keystrokes		Enter

Appendix G Participant 6 Level 1 Verbal Transcript and Coding

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
00:00.0				[E informed P that we were recording and he could ask any questions] Ok, I just go? [E tells yes that he can start]					
00:08.2	putty.exe	Windows Explorer	Running Applications						
00:10.6		login	usr name/psswd						
00:20.0	computer problem			Wait, is that it? It's not showing anything. [E seems perplexed and asks herself why it isn't doing anything]					
00:32.8			Ctrl c	[E says, ok there it is after typing Ctrl+c]	start				
00:37.0		unix_cmd = clear screen	clear	Ok, I'm just going to make a file.					
00:42.0		unix_cmd = make dir	mkdir test	Just a new directory so I don't, uh.					
00:48.6		unix_cmd = change dir	cd test						
00:52.4		unix_cmd = open file = nano, new file	nano test.c	Yeah, I'm just making a new directory so I don't have to uh just keep it separate from everything else. Then I'm going to make a file, um let's see. We'll just, I'll call it.					
	problem			Ok, now I'm in my program. Ok, first off I see that you're gonna have the square root in here, so you're gonna need math.h.	start: begin program			understand problem/plan (square root in prob needs math.h)	
01:16.0	putty.exe	txt_ed = typing	#include "math.h	So, I'll include that because otherwise it won't work.		start: include library files			write code: high prior knowl (include math lib)
01:23.8	Windows Explorer	L Button Down	FolderView						write code
01:24.2	Windows Explorer		Left Arrow	Oh no. [P is having trouble using the text editor]					write code
01:25.2	putty.exe	Windows Explorer	Running Applications						write code
01:26.6		txt_ed = move left, 6	Left Arrow						write code
01:28.0		txt_ed = delete	Backspace						write code
01:28.4		txt_ed = typing	<						write code
01:29.0		txt_ed = move right, 6	Right Arrow	I'll have to remember, I don't know which way it's going to be.				low prior knowl (include syntax)	write code
01:30.4		txt_ed = typing	>	We'll find out later.					write code
01:30.6		txt_ed = whitesp	Enter						write code
01:31.6		txt_ed = typing	#ic						write code
01:34.2		txt_ed = delete	Backspace						write code
01:34.4		txt_ed = typing	nclude <stdio.h>	Then uh, you'll probably need other include files. I don't know if that's c++ or c, but we'll find out.				low prior knowl (other include files)	write code: high prior knowl (include stdio lib)
01:41.2		txt_ed = whitesp	Enter	Uh, I get them confused.					write code
01:45.2		txt_ed = typing	#						write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
01:45.6		txt_ed = delete	Backspace	And that,					write code
01:48.8		txt_ed = move up	Up Arrow						write code
01:49.0		txt_ed = move right, 18	Right Arrow	actually it's uh,			incorrect detection: lib error	check-detect: inspect	diagnose (evaluate stdio.h as incorrect and use stdlib.h)
01:50.2		txt_ed = move left, 3	Left Arrow						fix code
01:51.2		txt_ed = delete, 2	Backspace						fix code
01:52.2		txt_ed = typing	lid						fix code
01:53.2		txt_ed = delete, 2	Backspace						fix code
01:54.6		txt_ed = typing	ib	that should be it, I think.		coded: include library files	incorrect fix/error: deletes needed library (stdio.h) adds unneeded library (stdlib.h)		fix code
01:55.8		txt_ed = move right, 3	Right Arrow						write code
01:56.6		txt_ed = whitesp	Enter						write code
01:56.8			Enter						write code
01:58.4		txt_ed = typing	int main	I'm going to start main, I guess. It's not going to return anything, but we'll just call it int because that is standard.		start: construct main routine			write code: high prior knowl (begin main subroutine; int main() {)
	problem			Ok, we're going to need three arrays, x, y, and z, and it looks like they will hold integers.				understand problem/plan (paraphrasing, three integer arrays)	
02:05.0	putty.exe	txt_ed = delete	Backspace						write code
02:05.2		txt_ed = typing	(){						write code
02:06.8		txt_ed = whitesp	Enter						write code
02:07.2		txt_ed = whitesp	Enter						write code
02:18.4		txt_ed = typing	int	Wait, no. Well, yeah. I'm reading it [referring to the problem]				understand problem/plan (read prob)	write code
	problem			Ok, yeah so, uh, if I remember how to do arrays right.	start: create/initialize data structures			low prior knowl (arrays)	
02:19.2	putty.exe	txt_ed = delete	Backspace						
02:41.6		txt_ed = typing	x[10] = {1, 2, 3, 4, 5};	And I just initialize them to anything? [E responds yes]		start: declare arrays; start: populate arrays		understand problem/plan (what to initialize arrays to)	use external source
				Ok, I'll just do 1 through 10 I guess. Or wait, they don't have to be completely full, they can just be anything. So, to save time.				understand problem/plan (what to initialize arrays to)	write code: high prior knowl (declare and init x & y array at same time)
03:02.6			Enter						write code
03:03.8		txt_ed = typing	int y						write code
03:05.6		txt_ed = delete	Backspace						write code
03:06.4		txt_ed = typing	[10] = {1, 2,	I haven't done arrays in so long. Wait let's get something different.				low prior knowl (arrays)	write code
03:14.0		txt_ed = delete, 6	Backspace						write code
03:16.0		txt_ed = typing	4, 5, 6,	So we can get something for 6.					write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
03:18.6		txt_ed = delete, 3	Backspace	Ah, that's not right.					write code
03:19.6		txt_ed = typing	6, 7, 8};	6, 7, 8. Ok,		coded: populate arrays	error: doesn't add sentinel value to x or y array		write code
03:25.4			Enter						write code
03:25.8			Enter	I have that and we're going to need one z, but it will be empty right now because z is. What is z?					write code
	problem			The product of x and y, so.				understand problem/plan (z has the product of x and y)	
03:38.0	putty.exe	txt_ed = typing	int						write code: high prior knowl (declare array; int z[10])
03:39.4		txt_ed = delete	Backspace						write code
03:39.8		txt_ed = typing	z						write code
03:41.6		txt_ed = delete	Backspace						write code
03:42.0		txt_ed = typing	[10];	That won't be anything yet. And then we need, uh let's see.		coded: declare arrays			write code
	problem			[E asks P what he is looking at] "Let n represent."		start: declare variable		understand problem/plan (read prob)	
				I don't know if we actually need n. I mean, well. Yes. Technically we do.				understand problem/plan (the purpose of n)	
04:08.0	putty.exe		Enter						
04:08.2			Enter						
04:09.0		txt_ed = typing	int n;	We'll put it in here. If we don't need it, we'll take it out later. [laughs]		coded: declare variable		understand problem/plan (put it in now & later take out)	write code: high prior knowl (declare n variable)
04:11.8			Enter						write code
04:12.0			Enter	Ok, so we're gonna need. Well, right now, since I made a n, I know that each one is gonna have		start: init variable			write code: high prior knowl (init n variable to number of actual elements)
04:21.2		txt_ed = delete, 3	Backspace	5 elements.					write code
04:25.8		txt_ed = typing	= 5;	So, n can go ahead and be 5.		coded: init variable			write code
04:29.2			Enter		coded: create/initialize data structures				write code
04:29.4			Enter	And then, we're going to need a loop to go through and calculate our uh, product.	start: compute/store product	start: construct loop			write code
04:36.8		txt_ed = typing	for	So,					write code
04:45.6		txt_ed = delete, 4	Backspace	let's, I don't like for loops.					write code: low prior knowl (avoids use of for loops)

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
04:47.4		txt_ed = typing	int i;	So, we're gonna make an i. i will be a counter.					write code: high prior knowl (need counter for while)
04:50.4			Enter						write code
04:50.6			Enter						write code
04:51.4		txt_ed = typing	while (i < 5	so uh, i is less than 5.			error: doesn't use n and hard codes loop ending value		write code: low prior knowl (while statement)
05:00.6		txt_ed = move left, 3	Left Arrow						write code
05:00.8		txt_ed = move right	Right Arrow						write code
05:01.4		txt_ed = delete	Backspace	Uh, actually 4.			incorrect detection: deletes correct loop index ending val (i < 5) add incorrect index ending val (i<4)	check-detect: inspect	diagnose (evaluate 5 as wrong and change to 4); fix code
05:01.6		txt_ed = typing	4)				error: only multiplies 4 of 5 values in the arrays		fix code
05:03.0			Up Arrow						write code
05:03.2		txt_ed = move left	Left Arrow						write code
05:03.4		txt_ed = move up	Up Arrow						write code
05:03.8		txt_ed = move down	Down Arrow						write code
05:04.0		txt_ed = move left	Left Arrow						write code
05:04.8			= 0	Because the way the arrays are counted, start at 0 and go to 4.					write code: high prior knowl (initialize loop counter; array indices start at 0)
05:07.0		txt_ed = move down, 3	Down Arrow	initialize i back to uh, 0. [P is referring to initializing the i before entering the loop, which is part of loop construction]					write code: high prior knowledge (initialize loop counter; array indices start at 0)
05:08.0		txt_ed = move up	Up Arrow						write code
05:09.0		txt_ed = move down	Down Arrow						write code
05:09.4		txt_ed = move right	Right Arrow						write code
05:11.0		txt_ed = delete, 2	Backspace						write code
05:12.0		txt_ed = typing	{	Then, um					write code
05:12.6			Enter						write code
05:12.8			Enter	we'll go calculate it. It will be x. If I remember this right.		start: multiply elements			write code
05:21.2		txt_ed = typing	x{	It's probably gonna be bad but,				low prior knowl (calculating product of arrays)	write code
05:26.0		txt_ed = delete	Backspace						write code
05:26.8		txt_ed = typing	[i	i, no we need z, z.					write code
05:28.0		txt_ed = delete, 3	Backspace						write code
05:29.0		txt_ed = typing	z[i] = x[i	z of i equals [pause]					write code: high prior knowl (z[i] gets x[i] times y[i] for the product)

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
05:38.2		txt_ed = delete	Backspace						write code
05:38.8		txt_ed = typing	] * y	i times y i.					write code
05:42.8		txt_ed = delete	Backspace						write code
05:43.0		txt_ed = typing	[i];			coded: multiply elements			write code
05:46.6			Enter						write code
05:47.6		txt_ed = typing	i++;	increase the counter.		revisit: construct loop			write code: high prior knowl (increment loop counter at end of loop)
05:51.0			Enter	and go and, next would be store the product, ok I've done that.				check-detect: inspect (code for product looks ok)	
05:51.8		txt_ed = typing	}		coded: compute/store product	coded: construct loop			write code
05:52.4			Enter	and then print all of them.	start: print arrays				write code
06:11.8		txt_ed = move down	Down Arrow	So, I could do the same thing.					write code
06:13.8		txt_ed = typing	while			start: construct loop			write code: high prior knowl (use another while)
06:14.8		txt_ed = delete	Backspace						write code
06:15.2		txt_ed = typing	(						write code
06:18.2		txt_ed = delete, 11	Backspace	Actually, we'll have to reinitialize the i back to, whoops that was too far, i back to zero.					write code: high prior knowl (initialize loop counter; array start at 0)
06:20.6			Enter						write code
06:20.8			Enter						write code
06:21.4		txt_ed = typing	}						write code
06:21.6			Enter						write code
06:22.4			Enter						write code
06:23.0		txt_ed = typing	i = 0;						write code
06:25.4			Enter						write code
06:25.8			Enter						write code
06:26.4		txt_ed = typing	while (i < 4){				error: doesn't use n and hard codes loop ending value		write code: low prior knowl (while statement)
06:34.6			Enter				error: only prints 4 of 5 values in the arrays		write code
06:35.0			Enter						write code
06:38.4		txt_ed = typing	fprintf(stdout, "%d	four, um, fprintf standard out [mumble] percent d. Well, we want something so people know what we are printing.		start: construct print statement			write code: high prior knowl (use text to identify printed numbers)
06:48.8		txt_ed = delete, 3	Backspace	Let's see [laugh].					write code
07:00.8		txt_ed = typing	Array	Well it's just going to be a random list of numbers, you can't just have um.					write code
07:03.8		txt_ed = move left, 5	Left Arrow						write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
07:04.8		txt_ed = delete	Backspace						write code
07:05.4		txt_ed = move right, 5	Right Arrow						write code
07:06.6		txt_ed = typing	X: Array	We'll call this array x, and we'll have array.					write code
07:12.4		txt_ed = delete, 6	Backspace	Let's put more spaces in there.					write code
07:13.4		txt_ed = typing	Array U	Array y.					write code
07:16.2		txt_ed = delete	Backspace						write code
07:16.4		txt_ed = typing	y						write code
07:17.0		txt_ed = delete	Backspace						write code
07:17.2		txt_ed = typing	Y: A	And then we'll have					write code
07:19.8		txt_ed = delete	Backspace						write code
07:20.0		txt_ed = typing	Arra	array, uh.					write code
07:22.4		txt_ed = delete, 4	Backspace						write code
07:23.6		txt_ed = typing	Pro						write code
07:24.6		txt_ed = delete	Backspace						write code
07:25.4		txt_ed = typing	oduct of X and Y	I forgot to do the square root, but we can do that later. This is going to be a huge line. X and y, and then, that's actually going to be z, but.					write code
07:50.2		txt_ed = delete, 20	Backspace	Let's change that, I don't like that [laughs].					write code
07:52.0		txt_ed = typing	Array XZ						write code
07:55.0		txt_ed = delete, 2	Backspace						write code
07:55.6		txt_ed = typing	Z (Product of X and While	Array z, and um. I was thinking ahead, that's what I did.					write code
08:02.4		txt_ed = delete, 5	Backspace						write code
08:04.0		txt_ed = typing	Y)						write code
08:06.6		txt_ed = move left, 24	Left Arrow						write code
08:08.0		txt_ed = move right, 3	Right Arrow						write code
08:09.6		txt_ed = typing	:						write code
08:09.6		L Button Down	FolderView						write code
08:10.2		txt_ed = move right	Right Arrow	The product of x and y.					
08:11.6		L Button Down	spider5.cs.clemson.edu - PuTTY						
08:13.0		txt_ed = move right, 21	Right Arrow				Error: Prints header inside loop		
08:15.8		txt_ed = typing	Square root of Z:");	and then we'll have square root of z. That's going to be horrible for formatting, but anyway.	start: square root of z; start: print square root				write code: low prior knowl (print header inside loop)
08:27.8			Enter						write code
08:28.2			Enter	ok, so we have that. Make sure we're doing that right,					write code
	problem			store the product [mumble] then print. [E reminds P to think aloud] Ok, I am reading the last two sentences. "Store the product."				understand problem/plan (read last 2 sentences)	
				Ok, I could do this all in one go, like, like I'm doing now I have. Which would be ok but it's not exactly what the directions say to do.				understand problem/plan	

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
				It says print contents of x, y, and z, then go.				understand problem/plan (order of print and square root)	
				But, I want to be fast so I want to go ahead and do this.				understand problem/plan	
09:08.4		txt_ed = typing	fprint	Do it my way.			error: incorrect fprintf, leaves off f		write code: high prior knowl (print array values)
09:09.8		txt_ed = delete	Backspace						write code
09:10.2		txt_ed = typing	(stdout,	So that it's the same end product.					write code
09:15.2		txt_ed = move up	Up Arrow	Oh, let's do a slash n,					write code
09:15.4			Up Arrow						write code
09:15.6		txt_ed = move right, 3	Right Arrow						write code
09:16.8		txt_ed = typing	\n	which is a newline. Make it look pretty.					write code
09:19.4		txt_ed = move down, 2	Down Arrow	Now, um,					write code
09:22.6		txt_ed = typing	"						write code
09:23.4		txt_ed = move left	Left Arrow						write code
09:23.6		txt_ed = typing	Spacebar						write code
09:23.8		txt_ed = move right	Right Arrow	we'll have					write code
09:26.0		txt_ed = typing	%d %d %d D	percent d. Wait no, we're gonna have to change that because if I leave it the way it is. Let me go ahead and finish this. Percent d, percent d, and then it will actually be. I think if I remember right.	revisit: print arrays			low prior knowl (syntax for fprint)	write code
09:47.8		txt_ed = delete	Backspace						write code
09:48.2		txt_ed = typing	%lf	L f, that would actually be uh, a floating point or double or whatever, which I want to be using.					write code
09:52.2		txt_ed = move left, 3	Left Arrow						write code
09:53.0		txt_ed = typing							write code
09:53.6		txt_ed = move right, 3	Right Arrow						write code
09:55.6		txt_ed = typing	", x[i], y	and this will be x of i, then y sub i,					write code
10:08.6		txt_ed = delete	Backspace						write code
10:09.0		txt_ed = typing	[i], x						write code
10:13.8		txt_ed = delete	Backspace	and then,					write code
10:14.0		txt_ed = typing	z						write code
10:14.8		txt_ed = delete, 2	Backspace						write code
10:19.4		txt_ed = typing	z[i], sqrtr	it will be uh, z sub i, and then right here it will be square root, I think that's right,	revisit: square root of sum of z; revisit: print square root	start: use square root function			write code
10:28.0		txt_ed = delete, 2	Backspace						write code
10:29.2		txt_ed = typing	t(z[i])	of z of i,	coded: square root of sum of z; coded: print square root	coded: use square root function	error: incorrect square root of individual items in z		write code
10:29.2		txt_ed = typing	);	and then. Print the z's, check. There, one more.	revisit: print arrays	revisit: construct print statement		check-detect: inspect	write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
10:45.2		txt_ed = move left, 33	Left Arrow	and then go back and put a newline.					write code
10:48.0		txt_ed = typing	\n	Otherwise it will look messy.		coded: construct print statement			write code
10:50.4		txt_ed = move up, 2	Up Arrow	Ok, um.					write code
10:55.6			Left Arrow	I'm going to need to cut this out because otherwise everytime it's going to print.			detected: header inside loop error	check-detect: inspect	diagnose (header in wrong place, inside loop)
10:56.0		txt_ed = move up, 2	Up Arrow						fix code
10:56.8		txt_ed = move down	Down Arrow						fix code
10:58.0		L Button Down	spider5.cs.clemson.edu - PuTTY						fix code
10:59.6		L Button Down	spider5.cs.clemson.edu - PuTTY						fix code
11:00.2		L Button Down	spider5.cs.clemson.edu - PuTTY	I don't know if it works this way, but we'll try it.					fix code
11:01.6		R Button Down	spider5.cs.clemson.edu - PuTTY	oh no, what happened? I see. Oooh, that was messy. Let's see, huh.					
11:13.0		txt_ed = delete	Backspace						
11:16.8			Enter	[P seems perplexed about cut and paste. E tells P that it just pasted it twice] That might be it.					
11:20.0		txt_ed = move down, 3	Down Arrow						
11:21.2		txt_ed = move up, 4	Up Arrow						
11:23.8		txt_ed = move down, 2	Down Arrow	Oh, ok.					
11:25.0		txt_ed = delete, 49	Backspace	Let's see.					fix code
11:36.8		txt_ed = move right, 22	Right Arrow						fix code
11:38.8		txt_ed = delete, 23	Backspace	Well, in that case we'll just uh, do the easy thing.					fix code
11:42.8		txt_ed = move left, 22	Left Arrow						fix code
11:45.0		txt_ed = delete	Backspace						fix code
11:45.8		txt_ed = typing	Spacebar						fix code
11:46.6			Spacebar						fix code
11:46.8		txt_ed = move left, 22	Left Arrow						fix code
11:49.4			Tab	Let's move that in.			error: breaks up a print statement w/ tab		fix code
11:49.8		txt_ed = move up, 3	Up Arrow						fix code
11:51.0		txt_ed = move right, 4	Right Arrow	We'll cut this out.					fix code
11:51.6		txt_ed = move left, 18	Left Arrow	I'm trying to think what that command would be.					fix code
11:53.0		txt_ed = move down	Down Arrow						fix code
11:53.2		txt_ed = move right	Right Arrow						fix code
11:53.4		txt_ed = move down	Down Arrow						fix code
11:53.8		txt_ed = move up	Up Arrow						fix code
11:54.8		Ctrl	k						fix code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
11:55.4		txt_ed = move down, 3	Down Arrow						fix code
11:56.6		Ctrl	Ctrl u	There we go.		finish: construct print statement	fixes: header inside loop error		write code
11:58.4		txt_ed = move down	Down Arrow						write code
11:59.2		txt_ed = move up, 2	Up Arrow	That was nice.					write code
12:00.8			Enter						write code
12:02.4		txt_ed = move down, 3	Down Arrow	Ok, we'll need a i plus, plus.		revisit: construct loop			write code: high prior knowl (increment loop counter at end of loop)
12:04.4		txt_ed = typing	i++				error: leaves off semicolon		write code: high prior knowl (increment loop counter at end of loop)
12:07.4		txt_ed = move left, 3	Left Arrow						write code
12:08.2			Enter						write code
12:08.6		txt_ed = move right, 3	Right Arrow						write code
12:09.8			Enter						write code
12:10.4			Enter						write code
12:11.4		txt_ed = typing	}	And then we'll do that, and.		coded:construct loop			write code
12:16.4		txt_ed = move up, 9	Up Arrow	Wait, what's going on here?				check-detect: inspect	
12:17.6		txt_ed = move right, 21	Right Arrow					check-detect: inspect	
12:21.6		txt_ed = delete, 5	Backspace						write code
12:24.0		txt_ed = typing	f						write code
12:26.6		txt_ed = move down, 12	Down Arrow	We need to close that.				check-detect: inspect (checking that brackets {} match)	
12:44.2		txt_ed = move down, 39	Down Arrow	Ok, I'm going to need to close.				check-detect: inspect (checking that brackets {} match)	
12:46.8		txt_ed = move up, 2	Up Arrow					check-detect: inspect (checking that brackets {} match)	
12:47.6		txt_ed = move down, 2	Down Arrow	I got that one closed.	coded: print arrays			check-detect: inspect (checking that brackets {} match)	
12:49.4			Enter						write code
12:49.8		txt_ed = typing	return 0;	even though we're just going to return zero just because c will forget if we don't.	revisit: begin program				write code: high prior knowl (return from main)
12:52.6			Enter						write code
12:52.8			Enter						write code
12:53.6		txt_ed = typing	}		coded: begin program	coded: construct main routine			write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
12:54.0		txt_ed = move up, 48	Up Arrow						
12:57.2		txt_ed = move down, 44	Down Arrow	and now I guess I can try it and run it, and see how bad I did at it.				low prior knowl (quality of the program)	
13:00.0		txt_ed = move up, 16	Up Arrow						
13:03.0		Ctrl	Ctrl o	write it out.					
13:04.4			Enter						
13:05.4		Ctrl	x						
13:06.6		unix_cmd = compile prog	gcc -Wall -lm test.c	gcc, wall for warnings, and because math.h is in there, you have to do lm otherwise it will freak out again.					check-detect: compiler
13:15.6			Enter	I'm probably going to get 300 warnings. Yea.					check-detect: compiler
				I left out an include. That's what I did.	revisit: begin program		detected: library error (left out stdio.h)		check-detect: compiler; diagnose (left out lib)
13:22.2		unix_cmd = open file	Up Arrow						
13:24.4		txt_ed = move down, 2	Down Arrow						
13:25.8		txt_ed = typing	#include<stdio.h>	uh, um s-t-i, standard io for all those.	coded: begin program	finish: include library files	fixes: adds library		fix code
13:35.4			Enter						
13:36.6		Ctrl	o						
13:37.2			Enter						
13:38.6		Ctrl	x	Now, I should probably should have checked the other errors and tried to fix them too, but					
13:39.6		unix_cmd = clear screen	Up Arrow	clear, make it easier.					
13:44.0			Enter						
13:44.6		unix_cmd = compile prog	Up Arrow						check-detect: compiler
13:46.0			Enter	ok, 25, 26, product undeclared, what? Ok, anyway.					check-detect: compiler
13:56.8		unix_cmd = compile prog	Up Arrow	[E asks P what he is reading] If I fix, usually when you get these type of compiler errors, if you get one that says missing something, a lot of the ones after it can be eliminated just because you left something out. It reads everything else incorrectly.					check-detect: compiler
14:11.2			Enter	Wow, I meant to open it, not compile it.					
14:12.4		unix_cmd = open file	Up Arrow						
14:14.4		txt_ed = move down, 12	Down Arrow	So, wherever line 25 is.					diagnose
14:17.0		Ctrl	c	that's 13.					diagnose
14:18.4		txt_ed = move down, 12	Down Arrow	get on down some more.					diagnose
14:21.0		Ctrl	Ctrl c	it's there. [P is on line 25]					diagnose
14:24.0		txt_ed = move down	Down Arrow						diagnose

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
14:24.6		txt_ed = move right	Right Arrow	Hmmm, [pause] let's see. Let me go back one more line it'll make it.			detected: print statement on two lines error		diagnose (on line 26 and need to backspace)
14:33.0		txt_ed = delete, 2	Backspace				fixes: print statement on two lines error		fix code
14:36.4		putty.exe = resize	spider5.cs.clemson.edu - PuTTY						
14:40.2		Ctrl	Ctrl c	Yeah, that's 25, so go that. We got that one. [E asks P to say what he is thinking] I'm trying to figure out, I'm just looking at the uh. Everything looks fine.				check-detect: inspect	
15:02.0		Ctrl	Ctrl o	Let me try it again just to be sure.					
15:03.2			Enter						
15:04.0		Ctrl	Ctrl x						
15:04.8		unix_cmd = clear screen	Up Arrow						
15:06.0			Enter						
15:06.4		unix_cmd = compile prog	Up Arrow						check-detect: compiler
15:07.4			Enter	See, just by moving those two move it around. Ok. I left out uh f on that. [P is referring to the compiler mssg about fprint]			detected: fprintf error		check-detect: compiler (less errors and error about print stmt); diagnose (left out f on print statement)
15:13.8		unix_cmd = open file	Up Arrow						
15:15.8		txt_ed = move down, 31	Down Arrow	somewhere, where is.					diagnose
15:18.8		txt_ed = move up, 3	Up Arrow						diagnose
15:19.4		txt_ed = move right, 6	Right Arrow	yeah.					diagnose
15:21.2		txt_ed = typing	f	that'll fix that which could fix the other ones.			fixes: fprinf error		fix code
15:22.2		Ctrl	Ctrl o						
15:22.6			Enter						
15:23.2		Ctrl	Ctrl x						
15:24.0		unix_cmd = compile prog	Up Arrow						check-detect: compiler
15:25.0			Enter	Ok, 33. [P is referring to a semicolon error] I never used n. [P referring to compiler mssg] Let's see, let's see why.			detected: didn't use n & line 33		check-detect: compiler (error on line 33 & didn't use n)
15:33.8		unix_cmd = open file	Up Arrow	But let me fix my 33 first. I missed something.					
15:39.8		txt_ed = move down	Down Arrow						diagnose
15:39.8		txt_ed = move right	Right Arrow						diagnose
15:41.0		txt_ed = move down, 40	Down Arrow	Go down.					diagnose

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
15:43.0		txt_ed = move up, 8	Up Arrow						diagnose
15:44.6		Ctrl	c	[P is on line 30]					diagnose
15:45.6		txt_ed = move down, 5	Down Arrow						diagnose
15:47.2		Ctrl	Ctrl c	[P is on line 35]					diagnose
15:49.6		txt_ed = move up, 12	Up Arrow	Wait, what did it.					diagnose
15:53.2		txt_ed = move down	Down Arrow						diagnose
15:53.4		txt_ed = move up, 10	Up Arrow						diagnose
15:54.8		txt_ed = move down	Down Arrow						diagnose
15:55.6		txt_ed = move right	Right Arrow						diagnose
15:56.0		txt_ed = move up, 3	Up Arrow						diagnose
15:57.2		txt_ed = move right, 4	Right Arrow						diagnose
15:58.6		txt_ed = move down, 2	Down Arrow						diagnose
15:59.2		txt_ed = move up, 2	Up Arrow	Well, I can use n but we don't need it cause this is all static and we weren't doing anything dynamic.					diagnose
16:00.0		txt_ed = move right, 4	Right Arrow						
16:01.2		txt_ed = delete, 11	Backspace				fixes: deletes n variable		fix code
16:03.4		Ctrl	o						
16:03.6			Enter						
16:04.6		Ctrl	x						
16:05.8		unix_cmd = compile prog	Up Arrow						check-detect: compiler
16:07.2			Enter	Um, ok, "expected semicolon before [mumble]" [P reads line 33 error again]					check-detect: compiler (expected semicolon)
16:12.6		unix_cmd = open file	Up Arrow						
16:13.8			Enter						
16:14.2		txt_ed = move down, 29	Down Arrow	Ok, counting brackets or braces or whatever they're called. That and that. Those two go together. Main is up there. That one and that one, and that.					diagnose
16:27.0		txt_ed = move right, 3	Right Arrow	oh right here.			detected: missing semicolon		diagnose
16:28.4		txt_ed = typing	;	That's what I needed.			fixes: added semicolon		fix code
16:29.6		Ctrl	Ctrl o						
16:29.8			Enter						
16:30.8		Ctrl	Ctrl x						
16:31.6		unix_cmd = compile prog	Up Arrow						check-detect: compiler
16:32.4			Enter	ok, it compiled.	finish: begin program	finsh: main construct; finish: declare arrays			check-detect: compiler
16:35.2		unix_cmd = run prog	./a.out	now, we try running it. That's half the battle.					check-detect: execute

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
16:37.2			Enter	Now, it didn't format correctly, but ok. [E informs P that the formatting isn't very important.] We got square root of 4 is 2, so that's right. Square root of 10, that's about right. That's about right, and that's about right.	finish: create/init data structures; finish: compute/store product in z; finish: print arrays				check-detect: execute (formatting and square root of elements)
17:01.2	Windows Explorer	L Button Down	start	If we want to be real technical, we can.					check-detect: execute (calculator)
17:02.6		Windows Explorer	L Button Down	[computer problems, E informs P that it is slow while recording] Let's see, um. There ok, so, we can try 10.					check-detect: execute (calculator)
17:29.8	Calculator	L Button Down	10						check-detect: execute (calculator)
17:31.2		L Button Down	sqt	yep, that's right. It rounded for us, how nice.					check-detect: execute (calculator)
17:37.2		L Button Down	C						check-detect: execute (calculator)
17:38.2		L Button Down	18	We get 18.					check-detect: execute (calculator)
17:39.2		L Button Down	sqt	yep.					check-detect: execute (calculator)
17:44.6		L Button Down	C						check-detect: execute (calculator)
17:45.8		L Button Down	28	then 28.					check-detect: execute (calculator)
17:47.4		L Button Down	sqt	yeah.					check-detect: execute (calculator)
17:51.2		L Button Down	Calculator						check-detect: execute (calculator)
17:52.4	putty.exe	Windows Explorer	Running Applications	So, I mean, if you want me to, I can go and fix the formatting?					diagnose (only formatting errors)
17:55.0		unix_cmd	Up Arrow	[E tells P no].					use external source
17:58.4		unix_cmd = clear screen	clear	Ok, [laughs] I mean we get graded for that, so. [Intresting that the output format gets graded but he doesn't worry about internal formatting, i.e. everything lined up on left making it hard to read]			all the code is lined up on the left but he is worried about output formatting		
17:59.4			Enter	[E asks P if he would say that he is done] Let me read through it, just to be sure, and make sure that I did everything.				check-detect: compare	
18:11.0	problem			[pause while reading] "populated by". Ok, so there's no dynamic input, so. It's all static if you do it from within the program. So, that means the me the programmer knows what I'm gonna get, so hence all good.				understand problem/plan (read prob); understand problem/plan (all static not dynamic in problem)	

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
				But we don't need it cause we already know.				check-detect: compare	
				change everything else, but.[E asks P what all he would have to change] Like, if you were to change instead of me just having i guess put 5 in each one and you were to put like 6 in each one, then you'd have to make your loop counter only count that many or if you had a different amount in each one, you'd have to. For the larger one, you'd have to count all the way up for it, and for the smaller one,				check-detect: inspect	
				you could do it separately or if you were ambitious, you could do them in the same.				understand problem/plan	
				But I'm not that ambitious, so I would do it the quickest and fastest way.				understand problem/plan (do the fastest)	
				"Store", yes I did that in a third array. Printed them all out, and then.				understand problem/plan (read prob); check-detect: compare	
				Wait, I read that wrong.					diagnose (read prob wrong)
19:35.2		unix_cmd = open file	Up Arrow	I found that.					
19:40.2		txt_ed = move down, 37	Down Arrow	ok.				check-detect: compare	
19:42.6		txt_ed = move up, 9	Up Arrow	So, we can get rid of, uh, not the square root of each item in z,			detected: square root of individual items in z		diagnose (not square root of each item, but the sum of items)
19:45.4		txt_ed = move down	Down Arrow						fix code
19:45.6		txt_ed = move up	Up Arrow						fix code
19:46.0		txt_ed = move right, 33	Right Arrow	but the sum of all the items in z.	revisit: compute square root of sum of z				fix code
19:49.2		txt_ed = move down	Down Arrow						fix code
19:49.4		txt_ed = move up	Up Arrow						fix code
19:49.6		txt_ed = move right, 39	Right Arrow						fix code
19:51.8			Left Arrow	I had a good program though.					fix code
19:53.2		txt_ed = move up, 3	Up Arrow						fix code
19:55.2			Left Arrow						fix code
19:56.4		txt_ed = delete, 24	Backspace	ok, let's go ahead and just change that.					fix code
19:59.8		txt_ed = typing	\n");						fix code
20:02.8		txt_ed = move down, 4	Down Arrow	Let's see,					fix code
20:06.6			Left Arrow						fix code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
20:06.8			Left Arrow						fix code
20:08.8		txt_ed = delete, 12	Backspace	we don't need this.			fixes: delete square root of each element		fix code
20:12.2		txt_ed = move down, 5	Down Arrow	I mean, I can make a separate loop, but that's just.			error: does not delete the %lf in print statement		write code: high prior knowl (need separate loop or not)
20:22.2			Enter	yeah, let's go ahead and do it.		start: construct loop			write code: high prior knowl (use another loop)
20:23.8		txt_ed = typing	while (i<	while to calculate the uh.					write code: low prior knowl (construct while loop)
20:28.4		txt_ed = move up, 24	Up Arrow	we're going to need a sum.	revisit: create/init data structures	start: declare other variables			write code: high prior knowl (need a sum variable)
20:30.0		txt_ed = move down, 6	Down Arrow						
20:31.2		txt_ed = move up, 8	Up Arrow	up here.					
20:32.8		txt_ed = move down, 5	Down Arrow						
20:35.4		txt_ed = typing	int sum;		coded: create/init data structures	coded: declare other variables			write code: high prior knowl (declare sum var)
20:39.2			Enter						write code
20:39.8		txt_ed = move down, 21	Down Arrow						
20:42.6		txt_ed = move right, 9	Right Arrow	ok, while i is less than 4 because we're gonna have 4 in there.	revisit: square root of sum of z	revisit: construct loop	error: there are 5 elements not 4		write code: low prior knowl (the need for 4 as the loop ending value)
20:45.4		txt_ed = typing	4)						write code
20:48.0		txt_ed = move up	Up Arrow						write code
20:48.8			Enter	We need to reinitialize i.					write code: high prior knowl (need to reinitialize I for while loop)
20:50.0		txt_ed = typing	i = 0;	I am all over the place.					write code: high prior knowl (need to reinitialize I for while loop)
20:53.4		txt_ed = move down	Down Arrow						
20:53.6		txt_ed = move right, 5	Right Arrow	Go to uh,					
20:56.0		txt_ed = typing	{	let's see.					write code: high prior knowl (open while stmt)
20:57.2			Enter						write code
20:57.4			Enter	We'll need uh,					write code
21:03.8		txt_ed = typing	sum = sum + z[i];	sum equals sum plus, it'll be z i.		start: calculate sum of z; coded: calculate sum of z			write code: high prior knowl (calculate the sum of values in z array)
21:17.8			Enter	and then,					write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
21:19.4		txt_ed = move up	Up Arrow					check-detect: inspect	
21:19.8		txt_ed = move down	Down Arrow					check-detect: inspect	
21:20.2		txt_ed = move up, 5	Up Arrow					check-detect: inspect	
21:21.4		txt_ed = move down, 5	Down Arrow					check-detect: inspect	
21:22.6		txt_ed = typing	}	close that.		coded: construct loop	error: does not increment the loop counter		write code: low prior knowl (close while stmt w/o incrementing loop counter)
21:23.0		txt_ed = move up, 27	Up Arrow						
21:24.8		txt_ed = move down	Down Arrow	But, to do that,					
21:25.0		txt_ed = move right, 6	Right Arrow	this will need to be a zero.	revisit: create/init variables	start: initialize other variables			write code: high prior knowl (need to initialize sum to zero)
21:26.4		txt_ed = typing	= 0		coded: create/init variables	coded: initialize other variables			write code: high prior knowl (need to initialize sum to zero)
21:28.6		txt_ed = move down, 28	Down Arrow	otherwise we'll get an incorrect value because sum could be anything on the first run, um.				check-detect: inspect	
21:30.8		txt_ed = move up	Up Arrow	That calculated the sum.		finish: calculate sum of z		check-detect: inspect	
21:38.6			Enter						write code
21:39.0			Enter						write code
21:40.0		txt_ed = move up	Up Arrow						write code
21:41.6		txt_ed = typing	fprt	Now we can just do uh, f	revisit: print square root of sum	start: construct print statement			write code: high prior knowl (construct a print statement)
21:45.4		txt_ed = delete	Backspace						write code
21:45.8		txt_ed = typing	intf(stdout, "%ld	printf out then it'll be percent l f.					write code
21:52.6		txt_ed = delete	Backspace						write code
21:52.6		txt_ed = typing	f"						write code
21:55.8		txt_ed = delete	Backspace						write code
21:56.6		txt_ed = typing	, Sw	and then um, that'll be the	revisit: square root of sum of z	start: use square root function	error: square root function is capital		write code: high prior knowl (use square root function)
21:59.0		txt_ed = delete	Backspace						write code
21:59.2		txt_ed = typing	qrto	square root					write code
22:00.8		txt_ed = delete	Backspace						write code
22:01.4		txt_ed = typing	(sum)	of sum.	coded: square root of sum of z;	coded: use square root function;			write code
		txt_ed = typing	);	Um, parenthesis again.	revisit: print square root of sum; coded: print square root of sum	revisit: construct print statement; coded: construct print statement		check-detect: inspect	write code
22:09.0		txt_ed = move up	Up Arrow						write code

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
22:09.6			Enter						write code
22:10.0		txt_ed = move down	Down Arrow						write code
22:10.8		Ctrl	Ctrl o						
22:11.2			Enter						
22:12.0		Ctrl	Ctrl x						
22:12.8		unix_cmd = compile prog	Up Arrow	Now we can try it, compiling it and see what.					check-detect: compiler
22:14.6			Enter	ok, that's weird. Ok, on. Ok. [E asks P what he is thinking is weird]					check-detect: compiler
22:29.0				I hardly ever get undefined symbols, but I'll just have to go look and see if I typed something wrong.					check-detect: compiler
		unix_cmd = open file	Up Arrow	"Too few arguments, line number 29" [P reading compiler error]					check-detect: compiler
22:44.4			Enter						diagnose
22:44.8		txt_ed = move down, 21	Down Arrow	So, let's go to line 29.					diagnose
22:46.8		Ctrl	Ctrl c						diagnose
22:48.0		txt_ed = move down, 8	Down Arrow						diagnose
22:50.0		txt_ed = move up	Up Arrow						diagnose
22:51.2		Ctrl	Ctrl c						diagnose
22:52.8		txt_ed = move right, 38	Right Arrow	oh, I took out. Still gotta take out that,			detected: %lf in print statement for arrays		diagnose (took out arg so need to take out the formatting for it)
22:55.0		txt_ed = delete, 7	Backspace	and do that.					fix code
22:58.0		txt_ed = move down, 12	Down Arrow				fixes: print statement for arrays		
23:01.8		txt_ed = move right	Right Arrow	Yeah, I think I did capital S.			detected: square root function		diagnose (capitial S used in square root)
23:02.0		txt_ed = move left, 9	Left Arrow						diagnose
23:03.2		txt_ed = move right	Right Arrow	It looks like capital S, I can't tell.					diagnose
23:04.4			Backspace						fix code
23:04.6			s	yep, it is capital S.			fixes: square root function		fix code
23:05.6		Ctrl	Ctrl o						
23:06.0			Enter	That would be the problem.	finish: square root of sum of z.	finish: use square root function			
23:06.8		Ctrl	Ctrl x						
23:07.6		unix_cmd = compile prog	Up Arrow						check-detect: compiler
23:09.0			Enter	ok.					check-detect: compiler
23:09.8		unix_cmd = run prog	Up Arrow	now we'll try running it again.					check-detect: execute
23:12.8			Enter	and I'm not getting a print of.					check-detect: execute (no print of square root)

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
23:18.0			Enter	ok, that's weird, let's see.					check-detect: execute
23:25.0		Ctrl	Ctrl c						check-detect: execute
23:27.8		unix_cmd = open file	Up Arrow	It looked like an endless loop or something going on.					diagnose (endless loop)
23:30.2			Enter						
23:30.4		txt_ed = move down, 44	Down Arrow	oh, it would help if I uh,					diagnose
23:36.6		txt_ed = move up, 7	Up Arrow	increase the counter in here somewhere.			detected: did not increment loop counter		diagnose
23:38.6		txt_ed = move down	Down Arrow	[laughs]					fix code
23:39.0			Enter						fix code
23:39.4		txt_ed = move up	Up Arrow						fix code
23:40.8		txt_ed = typing	i +	otherwise it is always going to be less than 4.				check-detect: inspect	fix code
23:45.0			Backspace						fix code
23:45.0			Backspace						fix code
23:45.6		txt_ed = typing	++;						fix code
23:47.8		txt_ed = move up	Up Arrow				fixes: increment loop counter		fix code
23:48.0		txt_ed = move right, 13	Right Arrow						
23:49.4			Enter						
23:50.2		Ctrl	Ctrl o						
23:50.6			Enter	there we go.					
23:51.2		Ctrl	Ctrl x						
23:51.8		unix_cmd = compile prog	Up Arrow						check-detect: compiler
23:53.4			Enter						check-detect: compiler
23:54.0		unix_cmd = run prog	Up Arrow						check-detect: execute
23:58.4			Enter	um, yeah that.					check-detect: execute
24:02.0			Up Arrow						check-detect: execute
24:02.4			Up Arrow						check-detect: execute
24:05.0	Windows Explorer	L Button Down	start	Let me open up, awe this is going to take forever again [slow opening of calculator].					check-detect: execute (calculator)
24:06.2		Windows Explorer	L Button Down	ok, let's see.					check-detect: execute (calculator)
24:29.0	Calculator	L Button Down	4	4					check-detect: execute (calculator)
24:30.0		L Button Down	+	plus					check-detect: execute (calculator)
24:30.8		L Button Down	10	10					check-detect: execute (calculator)

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
24:32.0		L Button Down	+	plus					check-detect: execute (calculator)
24:32.8		L Button Down	18	18					check-detect: execute (calculator)
24:34.2		L Button Down	+	plus					check-detect: execute (calculator)
24:34.6		L Button Down	28	28					check-detect: execute (calculator)
24:35.6		L Button Down	=						check-detect: execute (calculator)
24:37.0		L Button Down	sqt	square root					check-detect: execute (calculator)
24:41.6		L Button Down	C	yep.	finish: print square root				check-detect: execute
24:42.2		L Button Down	Calculator						
24:45.4	putty.exe	L Button Down	spider5.cs.clemson.edu - PuTTY	ok, and I could've got it wrong again.				check-detect: compare	
24:46.2		unix_cmd = clear screen	Up Arrow	So, I might wanna read it through again just to make sure.				check-detect: compare	
24:49.0			Enter						
24:51.0	problem			Ok. [P is rereading the problem statement] Yes, ok. [P is continuing to read the problem] That's weird though, the directions. I just realized something. If, let's say it says each can hold up to 10, but the two arrays can contain 5 actual. I don't know if I could do anything about this because I didn't write the directions. I'll just complain, but um, if you had one that is 3 elements long and the other one is 5 elements long. It says "store the product of corresponding elements", once you get to the 4th one, there's nothing there for the 3rd one. I don't want to deal with that because I don't know it well enough to say hey. I don't arrays well enough to say what happens when you try to multiply the 4th element by the 3rd that is non-existent but you can't say zero. [E reminds P that the problem statement says that the two are of the same length].				understand problem/plan (read prob); check-detect: compare; understand problem/plan (what about uneven arrays)	use external source

Elapsed Tm	Computer / Paper Page	User Computer Actions	Keystrokes/Detail	Verbal Transcript	Problem Goals	Subtask of Goal	Making and Fixing Errors	Metacognitive Strategies	Programming Strategies
				So then they are both going to be square.				understand problem/plan (arrays always even)	
26:46.6	finish	L Button Down	spider5.cs.clemson.edu - PuTTY	Then I think I am done.	finish		error: no sentinel value in arrays	check-detect: compare	

Appendix H Final Coding Document

Coding Definitions for Test of Computational Thinking research study As of: 5/23/2009

Accepted abbreviations:
E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action

Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page

URL of current web page
Title of current book
putty.exe = code this when user is interacting with the ssh terminal
problem page = code this when user is interacting with the piece of paper with question to answer
design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions

can code > 1 on one line

compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]
readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)
bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)
designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions

login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command
- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action
- **copy** = Copy text; describe the text being copied
- **cut** = Cut text; describe the text being cut
- **paste** = Paste text
- **move** = Moving the cursor around in the text editor; specify up, down, left, right and how many times.
- **whitesp** = Creating newlines/white space in the program for readability
- **delete** = Delete text by backspacing; describe the text that is being deleted
- **typing** = Typing text in the editor; (The text is captured in column E)
- **exit** = Exit the text editor;
- **save** = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action
- **brwwith** = Browsing within a content site; clicking a link on a content site that leads to another page in content site

brwbetw = Browsing between content sites = clicking a link on a content site that leads to a new content site.
goseng = Going to a search engine from the problem start page
sengsearch = Doing a search engine search
sengresult = Clicking on a search engine result.
sengresultnext = Browsing to the next (or previous) page of search engine results.
sitesearch = Doing a site search (ie, a within-site search engine search)
typurl = Typing a URL into the browser address bar.
back = Clicking BACK button
mouseover = Mousing over popup and cascading menus or over link text or headers.
copy = Copying info from a web site
scan = Scanning a page
scroll = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;
may do a small bit of paraphrasing as long as you keep all the original meaning;
when in doubt, transcribe every word.
eg, when P is reading aloud from the problem page, do **NOT** code **~~(reading prob page)~~**, **do code the words P is saying** Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don't know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel? Is a sentinal vaue just like null or what? [E "You can use null as an ending, umhum, it is a terminating value"] ok	consider prob (what is the sentinel value) using external info: E consider prob (confusion re sentinel value)

Good segmenting: because you DO know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel?	consider prob (what is the sentinel value)
Excel line 2:		Is a sentinal vaue just like null or what?	consider prop (confusion re sentinel value)
Excel line 3:		[E "You can use null as an ending, umhum, it is a terminating value"] ok	using external info: E

Integrate info from other sources (eg experimenter's notes, experimenter's words) when needed to understand participant's verbal comments. If P says "This sentence is confusing" and the notes say "P pointed to 2nd sentence of problem page", include this comment in the transcript, in brackets.

Formatting

Put participant's words in without brackets.
Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**
If P is reading from the problem text or a website put in quotes, eg, **"**Let n represent the actual number of data values**."**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, This sentence is confusing [P pointed to 3rd sentence of page]

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G & H Goals (see problem statement w/ goals and subtasks)

Background: The goals will mostly refer to user statements about the problem, and the subtasks will mostly refer to domain-specific tasks to complete goals from the problem (see the problem statement). For this example, the goals will refer to concepts from the problem; whereas, the subtasks will refer to arrays, loops, products, summing, square root, etc.

Code the goals in column G and the subtasks of goals in column H. In addition, code the start, revisit, coding, and finish of the goals/subtasks. Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action
  Eg, **"I am going to start with main",** code **start: begin program** in column G, code **start: construct main** in column H
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**; **eg**, **[P is typing int main(void) {],** code **start: begin program** in column G, code **start: construct main** in column H
- **P leaves** a goal or sub-goal/task in working memory and then **returns** to the goal/sub-goal/task – also code what he/she is revisiting, **eg**, **revisit: begin program** in column G, code **revisit: construct main** in column H
- P finishes writing code for a particular goal or subtask. This is coded in red; **eg**, code **coded: begin program** in column G, code **coded: construct main** in column H
- P finishes answering a particular goal or subtask after testing and validating to evaluate as ok. This is coded in red; **eg**, code **finish: begin program** in column G, code **finish: construct main** in column H

Eg of: Goals explicitly mentioned by user

Eg 1: user comment="Now I am going to compute and print the square root of the sum of z."
Code=start: compute and print square root in column G

Eg of: Goal/Subtask not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 2: user comment = "If we do the square root then we probably need math." Then the user types
"#include <math.h>"
From the user's words and keystrokes, you can **infer** a goal and subtask.
Code=start: include library files in column H

Note: There is no need to code the specific details about the goal or answer, as well as any observations made by E, because errors and content knowledge are coded in columns I & K.

Col I Making and fixing errors

Background: These errors can include syntax, semantic, and suboptimal errors.

Code whenever:

- P makes a syntax error in the subtask causing compiler errors; Eg, "that should be it, I think." [P changed stdio.h to stdlib.h], code as error: deletes needed library (stdio.h) adds unneeded library (include stdlib.h)
- P makes a logical error causing incorrect output; Eg, "Uh, actually 4." [P changed i<5 to i<4 for 5 elements], code as error: deletes correct loop index ending val (i < 5) add incorrect index ending val (i<4)
- P makes a suboptimal error that doesn't adversely affect the solution; Eg, error: Prints header inside loop
- P makes a goal/subtask error; Eg, error: incorrect square root of individual items in z

Code the error, when the error is detected, and when the error is fixed in red in column I.

When an error is introduced during coding, code error:
description
When an error is detected, code detected:
description
When P detects what P thinks is an error, but it is actually not, code incorrect
detection: description
When an error is fixed, code fixes:
description
When an error is introduced while not fixing another error, code incorrect
fix/error: description
When a fix to an error has been tested and validated as being fixed done fixing:
description

Cols J & K Metacognitive & Programming Strategies/Schemata can code > 1 on one line

Background: Metacognition (MC) – In Col J, code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem.

OR

Background: Schema (S) – In Col K, code whenever P makes a statement about his/her own content knowledge or makes a statement about a domain strategy used in the development of an answer, and these statements may be a written or a verbal response to a computational problem.

1. **Understand the Problem/Plan (MC)** code whenever P thinks about the problem statement**,** explicitly states needing to read or re-read the problem, reads or re-reads the problem statement or part of it out loud (also note what part of problem read), or asks a question/makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing** or questions about the **next task in the problem**). **Note**: If P is comparing his/her code to the problem statement, then this is coded as **check-detect: compare.** Can also code whenever P discusses general scheduling of activities, how he/she will allocate his/her time, how his/her high-level decision will affect multiple parts of the solution, or the needs and conditions to meet for specific tasks in the solution.

    eg, "Sentinel, what is a sentinel?", could code: **understand problem/plan (confusion re sentinel value)**
    eg, "and now I need to print them out, right. Let's see, where is it? [P re-reads the problem]", could code: **understand problem/plan (determine if next goal is to print out arrays)**
    eg, "First I have to sum the elements in z, then I can take the square root.", could code: **understand problem/plan**
    eg, "I'm gonna hard code 5 elements in the arrays and get that working, then I'll go back and make it more portable.", could code: **understand problem/plan**
    eg, "I will name them exactly what they are in the program, uh in the problem.", could code: **understand problem/plan**
    eg, "We'll put it in here. If we don't need it, we'll take it out later.", could code: **understand problem/plan**
    eg, "If we do the square root then we probably need math.", could code: **understand problem/plan (square root in prob needs math.h)**

2. **Design** (S) code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.
    eg, P writes pseudo-code for creating arrays in the solution to a problem before writing the computer code, code as **design (create arrays pseudo-code)**

    read/consider design (MC) code when P reads information from the piece of paper containing his/her design, drawings, and/or notes (also note what was read from the design) or P asks a question/makes a statement about his/her design

3. **Write code** (S) code when P writes code or mentions wanting to write code for the solution, **eg**, **write code:** - also code the details whenever P searches memory for relevant prior knowledge, makes statements that use his/her own prior knowledge including general

statements about the code, considering options for answer, self-questioning/ explaining statements about one's code, or low-level planning statements about prior content knowledge – also code whether the prior knowledge was high or low based on the correctness as well as the content knowledge.

[high|low] prior knowl (S) code when P's code (w/ or w/o words) shows accurate use of prior knowledge (and note the knowledge showing evidence of having, especially for loops & arrays) or shows inaccurate use/errors in prior knowledge. **Note**: The low prior knowl is similar to the errors coded in column I; however, this code may include more detail when the prior knowledge is not a syntax or semantic error but is low domain knowledge, such as using a while loop when a for loop is more appropriate.

eg "I just want to make it initialized by just go up by one, but, um", could code as **high prior knowl (initialize array to contiguous numbers using loop)**

eg "z sub i equals x sub i plus y sub i", could code as **high prior knowl (compute/store product of 2 arrays)**

eg "We can either use a for loop or a while loop to initialize x and y [P writes for loop]", could code as **high prior knowl (chooses for loop to init)**

eg "I can use a for loop or a while loop to calculate the sum. [P writes a while loop]", could code as **low prior knowl (chooses while loop instead of a for loop)**

eg "Ok, done with the loop. [P forgets to incr loop]", could code as **low prior knowl (P closes loop w/o incrementing loop counter)**

low prior knowl (MC) code whenever P's verbalization (not code) explicitly states needing to learn or find out more about a particular topic, i.e. help-seeking behavior, makes a clear low confidence statement about his/her answer, mentions the difficulty of some part of their task, or explicitly states guessing the computer syntax and using the compiler to determine correctness. **Note:** If P asks a question regarding the problem statement or mentions the difficulty of part of the problem/question, code as **understand problem/plan.**

eg, "what is the syntax for filling up an array?" [P uses the web to find out the syntax], code as **low prior knowl (array initialization syntax)**.

eg, "I really need to learn how to initialize arrays when I declare them", code as **low prior knowl (array initialization at time of declaration)**.

eg, "I know I should use a for loop, but I'm not good at for loops", **low prior knowl (for loop)**

eg, "this is the same thing I had trouble with a second ago", code as **low prior knowl (using arrays inside a loop)**

eg, "probably easier to just test it out and see if, and see what works", code as **low prior knowl (guessing then compile)**

4. **Check-Detect** (S & MC) code whenever P considers his/her answer or performs the act of testing, validating, or detecting errors (or lack of) in the solution. **This includes when P:**

inspect (MC) code when P checks code via visual/mental inspection and compares it to own knowledge and/or uses his/her own knowledge to execute existing code mentally, i.e. desk-checking. This includes when P reads computer code he/she has written to check the code or detect errors, evaluates/considers whether some information (eg, a partial answer) is a complete or correct answer, or quickly notices a logical error (other than typing or editor errors) and quickly changes his/her answer. **Note:** If P is comparing his/her answer to the problem statement, code this as **check-detect: compare**.

eg, "Print the z's, check. There, one more." [P is reading coding and matching parenthesis]; could code **inspect (reading print statement & matching parens)**

eg, P is scanning over his/her loop and evaluating whether the loop is correct or complete; could code **inspect (evaluating re loop)**

eg, "Actually, it's uh" [P changes loop from i<5 to i<4]; could code **inspect (loop stopping value)**

eg, "that equals one, then that will go to the second thing in the list equals one, so everything in the list will equal one and then,", code as **inspect (predicting list contents)**

compare (MC) code whenever P compares his/her code or knowledge to the problem statement. P could be looking either at the problem statement or at his/her answer when comparing.

eg, "Yep, I took the square root then printed it"; could code **compare (read prob and mentally inspect sol)**

eg, "ok, and I have completed summing the items, now what?"; could code **compare (decide if solution meets prob stmt)**

compile (S) code whenever P tests code via compiling it and inspecting the compiler error messages**.** This also includes when P reads information from the compiler or asks a question or makes a statement about the compiler message. **Note**: P is NOT looking at their code, only the compiler message**, and** if P is considering his/her answer after detecting an error, then code as **diagnose**

eg, P compiles code after writing some code; could code **check-detect: compile**

eg, "uh, for loop [referring to the error message given by the compiler]", could code as **compile (evaluate for loop as error)**

execute (S) code whenever P tests or mentions testing by executing a compiled program and compares the actual and expected output or uses test-write-statements or a debugging application/environment to inspect internal program data. This also includes when P explicitly reads information from the output after executing a program or asks a question or makes a statement about the program output by comparing the output to expected output/goals. **Note**: P is NOT looking at their code, only the program output**, and if** P is considering his/her answer after detecting an error, then code as **diagnose**

eg, P executes code after successfully compiling; could code **check-detect: execute**

eg, P inserts a print statement to check the sum of array elements before taking the square root of the sum, as specified in the problem; could code **execute (test-write for sum variable)**

eg, "Let me use the calculator to check my answers"; could code **execute (compare output to calculator)**

eg, "good, ok, just what I wanted [output from program x is printed with the values 1 - 10]", could code as **execute (output matched expected)**

5. **Diagnose** (S) code whenever P uses reasoning strategies to identify the cause of incorrect output/error or uses statements identifying the cause of an error without explicitly stating the reasoning. This includes when P poses possible questions and answers about the error he/she is diagnosing , checks code via visual/mental inspection and compares it to a compiler message or program output giving an error, or uses a specific approach (backtracking, recognition, elimination, etc.) for finding an error. **Note**: P must detect an error using one of the above methods, before using this code.

   eg, P is scanning over his/her loop and determining the possible error by comparing the code and a compiler message; could code **diagnose (inspecting compiler mssg and evaluating loop)**

   eg, P is scanning over his/her loop and determining the possible error by comparing the code and the output that was incorrect; could code **diagnose (inspecting program output and evaluating print loop)**

   eg. "I left out the include statement" [P automatically fixes error w/o inspecting], code as **diagnose (recognize left out stdio.h)**

   eg, "The x and y variables have the correct value, so it must be the z variable causing problems", could code as **diagnose (eliminate problem to z array)**

   eg, "It might be the type of brackets I used in the array declaration causing the error or it could be a brace I'm missing in one of my loops", could code as **diagnose (hypothesize about brackets in array creation or loops)**

6. **Fix code** (S) code whenever P changes code after detecting and diagnosing an error using one of the above methods. **Eg**, [P deletes stdlib.h in program and adds stdio.h], could code as **fix code**

Use external source: [book | web | old prog | E] code in Col K whenever P uses an external source of information to help with understanding the problem or answering the problem. In addition, code where the information is coming from, i.e. book, internet, old program, or E. **Eg**, P uses the experimenter to help with a problem, code as **using external source: E.**

Appendix I Version 1 of The Coding Document

Coding Definitions for Test of Computational Thinking research study As of: 1/21/2009

Accepted abbreviations:
E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action

Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page

URL of current web page
Title of current book
putty.exe = code this when user is interacting with the ssh terminal
problem page = code this when user is interacting with the piece of paper with question to answer
design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions

can code > 1 on one line

Note: when P answers a particular goal or subtask, this is coded in red. eg, answer goal 1, sub 1 (test.c program)
compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]
readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)
bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)
designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions

login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command
- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action

copy = Copy text; describe the text being copied

cut = Cut text; describe the text being cut

paste = Paste text

move = Moving the cursor around in the text editor; specify up, down, left, right and how many times.

whitesp = Creating newlines/white space in the program for readability

delete = Delete text by backspacing; describe the text that is being deleted

typing = Typing text in the editor; (The text is captured in column E)

exit = Exit the text editor;

save = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action

brwwith = Browsing within a content site; clicking a link on a content site that leads to another page in content site

brwbetw = Browsing between content sites = clicking a link on a content site that leads to a new content site.

goseng = Going to a search engine from the problem start page

sengsearch = Doing a search engine search

sengresult = Clicking on a search engine result.

sengresultnext = Browsing to the next (or previous) page of search engine results.

sitesearch = Doing a site search (ie, a within-site search engine search)

typurl = Typing a URL into the browser address bar.

back = Clicking BACK button

mouseover = Mousing over popup and cascading menus or over link text or headers.

copy = Copying info from a web site

scan = Scanning a page

scroll = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;

may do a small bit of paraphrasing as long as you keep all the original meaning;

when in doubt, transcribe every word.

eg, when P is reading aloud from the problem page, do **NOT** code **~~(reading prob page)~~**, **do code the words P is saying** Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don't know what statements codes refer to

Action Transcript Strategies/Metacognition Revised
Bloom
Excel line 1: Sentinel, what is a sentinel? need to learn (sentinel value) 1.1
Remember: Recalling,
Is a sentinal vaue just like null consider prob or answer D.a
Metacognitive: Strategic
or what? [E "You can use null (confusion re sentinel value) 2.1
Understand: Interpreting,
as an ending, umhum, it is a D.a
Metacognitive: Strategic
terminating value"] ok
External Source: Exp

Good segmenting: because you DO know what statements codes refer to

Action Transcript Strategies/Metacognition Revised
Bloom
Excel line 1: Sentinel, what is a sentinel? need to learn (sentinel value) 1.1
Remember: Recalling,
D.a
Metacognitive: Strategic
Excel line 2: Is a sentinal vaue just like null consider prob or answer 2.1
Understand: Interpreting, or what?
(confusion re sentinel value) D.a Metacognitive: Strategic
Excel line 3: [E "You can use null as an
External Source: Exp
ending, umhum, it is a
terminating value"] ok

Integrate info from other sources (eg experimenter's notes, experimenter's words) when needed to understand participant's verbal comments. If P says "This sentence is confusing" and the notes say "P pointed to 2nd sentence of problem page", include this comment in the transcript, in brackets.

Formatting

Put participant's words in without brackets.

Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**

If P is reading from the problem text or a website put in quotes, eg, **"**Let n represent the actual number of data values**."**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, This sentence is confusing [P pointed to 3rd sentence of page]

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G User goals (see problem statement w/ goals and subtasks)

Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action; **OR**
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**

These goals will mostly refer to user statement about the problem (see the problem statement), so they will refer to arrays, loops, products, summing, square root, and other concepts from the problem. See examples below.

Eg of: Goals explicitly mentioned by user

Eg 1: user comment="I am going to re-read the problem statement"
Code="re-read the problem"

Eg 2: user comment="Now I am going to compute and print the square root of the sum of z."
Code="answer question; goal 2, subtask 1"

Eg of: Goals not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 4: user comment = "If we do the square root then we probably need math." Then the user types
"#include <math.h>"
From the user's words and keystrokes, you can **infer** a goal and subtask.
Code="answer question; goal 5, subtask 1"

Col H Strategies / Metacognition details

Background: code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem. Can code > 1.
Codes:

read prob code when P reads or re-reads the problem statement or part of it; also note what part of problem read, eg, **read prob (1st sentence)**

draw/design code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.

read design code when P reads information from the piece of paper containing his/her design, drawings, and/or notes; also note what was read from the design.

read prog code when P reads computer code he/she has written; also note what was read from the program.

consider prob or answer code when the participant:

- asks a question or makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing**)

eg, "Sentinel, what is a sentinel?", could code: **consider prob or answer (prob confusion re sentinel value)**

- **OR** is evaluating or considering whether some information (eg, a partial answer) is a **complete or correct answer** (these could include statements by P of **low or high confidence in his/her answer**)

eg, P evaluating whether his loop is correct or complete & says "I know I should use a for loop, but I don't like for loops"; could code **consider prob or answer (evaluating answer re low confidence in for loop)**

Note: This includes judgment of knowing, feeling of knowing, monitoring progress toward goals, task difficulty, etc.

paraphrasing code whenever P **summarizes** the gist of some information coming from E, the web, or a book. Note: if P summarizes part of the problem or answer, code this as **consider prob or answer (paraphrasing prob)**, NOT just **paraphrasing**

task difficulty code whenever P mentions the difficulty of some part of their task. However, if P mentions the difficulty of part of the problem/question, code this under **consider prob or answer**,

eg, "this is the same thing I had trouble with a second ago", code as **task difficulty**

support code when P mentions that 2+ pieces of knowledge **support** or **agree with** or **confirm** or **provide evidence for** each other– also code the 2 pieces of knowledge. Note: this is a positive content evaluation.

eg, P retrieves a potential piece of code from own knowledge but says she needs to verify it; P searches and finds code in a book; then P enters this answer (or accepts previously entered answer as ok). The verifying info in the book would be coded as **support**-ing the potential answer from own knowledge

lack of support code when P mentions a **contradiction** or **disagreement** or **lack of support** between 2 or more pieces of knowledge – also code the 2 pieces of knowledge in parentheses. Note: this is a negative content evaluation.

need to learn code when the participant states needing to learn or find out more about a particular topic & uses an external source of information to answer, i.e. help-seeking behavior, and code the topic in parentheses if unclear, **eg**, "what is the syntax for filling up an array?", code as **need to learn(array initialization syntax)**.

self-question/answer code whenever the person **poses a question** to him/herself **concerning an external source** of information (a question that is not part of the problem or answer) or gives an **explanation/answer to a self-posed question** (code **self-answer** when P answers self due to a self-question or prior knowledge activation search)

prior knowledge activation code whenever P searches memory for relevant prior knowledge either before performing a task or during task performance, Note: this includes self-questioning statements about one's answer, goals, sub-goals, etc. **eg**, "I just want to make it initialized by just go up by one, but, um", code as **prior knowledge activation**.

site credibility code whenever P refers to the **credibility** or **believability** or **quality** or **accuracy** of the current website, or mentions **high or low confidence or trust** in the info on the website.

planning code whenever P discusses scheduling of activities or how he/she will allocate his/her time and effort. Note: planning includes goal statements about operations that are possible, postponed, or intended.

revisit goal code whenever P **restates** a goal or sub-goal in working memory – also code what he/she is recycling in parenthesis, eg, **revisit goal (need two arrays, Goal 2, Subtask 1)**

recycle goal code whenever P **reuses** a goal or sub-goal in working memory – also code what he/she is recycling in parenthesis.

hypothesizing code when P asks questions that go beyond the information presented in the problem, in the answer, and/or in an external source.

knowledge elaboration code whenever P elaborates on the information read, seen, or heard using his/her prior knowledge – also code the knowledge that is being elaborated in parenthesis,

eg, "ok, so, i is zero, put in numbers because that helps", code as **knowledge elaboration (evaluation strategy for answer)**

interest statement code whenever P mentions some part of their task is interesting – also indicate whether the statement was positive or negative, as well as the task of interest.

eg, "no, for loop again, god", code as **interest statement (negative for loops)**

Col I Revised Blooms Taxonomy (processes; knowledge)

Background: code whenever P verbalizes a complete thought or enough verbalization to infer meaning using P's computer actions – indicate processes, sub-processes, knowledge, and sub-knowledge.

Note: Terms and definitions directly taken from *A taxonomy for learning, teaching and assessing: A revision of Bloom's Taxonomy of educational objectives*.

eg, "how do I initial, uh, urrr, make them in a loop and add each other?", code as **1.1 Remember: Recognizing; C.a Procedural: Skills/Algorithms**

Processes

1. **Remember** – retrieve relevant knowledge from long-term memory

   1.1. Recognizing – locating knowledge in long-term memory that is consistent with presented material

   Example: Look for specific schema in memory needed for solution to a problem.

   Quote: "how do I initial, uh, urrr, make them in a loop and add each other."

   1.2. Recalling – retrieving relevant knowledge from long-term memory

   Example: Describe the schema located from long-term memory.

   Quote: "alright, start off with main file"

2. **Understand** – construct meaning from instructional messages, including oral, written, and graphic communication

   2.1. Interpreting – changing from one form of representation (e.g., numerical) to another (e.g., verbal) (e.g., paraphrase important speeches and documentation)

Example: Recognize the essential and nonessential parts of a problem statement.
Quote: “It says new data structure though.”

2.2. Exemplifying – finding a specific example or illustration of a concept or principle (e.g., give examples of various artistic painting styles)

Example: Provide/Use a specific example of the problem or answer to aid in understanding.
Quote: “ok, so, i is zero, put in numbers because that helps.”

2.3. Classifying – determining that something belongs to a category (e.g., concept or principle) (e.g., classify observed or described cases of mental disorders)

Example: Recognize how a data structure applies to many problems or visa versa.
Quote: “Ok so I was thinking either you have an array or a linked list of some sort…”

2.4. Summarizing – abstracting a general theme or major point(s) (e.g., write a short summary of the events portrayed on a videotape)

Example: Paraphrasing the problem statement in one’s own words.
Quote: “ok I just need to store the products in z.”

2.5. Inferring – drawing a logical conclusion from presented information (e.g., in learning a foreign language, infer grammatical principles from examples)

Example: Determine the skills/algorithm to use for a problem based on an example from a similar problem.
Quote: “alright, based on a similar problem I want to add in this problem”

2.6. Comparing – detecting correspondences between two ideas, objects, and the like (e.g., compare historical events to contemporary situations)

Example: Use two programs to determine how they are similar.
Quote: “and I'm going through that old code to compare to this code”

2.7. Explaining – constructing a cause-and-effect model of a system (e.g., Explain the causes of important 18th century events in France) **Note**: this may include low-level planning statements.

Example: Provide the relationship between operations in a procedure affected by each other.
Quote: “we're just going to make the first value initialed to something, then I can do a plus plus”

3. **Apply** – carry out or use a procedure in a given situation

3.1. Executing – applying a procedure to a familiar task (e.g., divide one whole number by another whole number, both with multiple digits) **Note**: usually execution is with the intent to analyze or evaluate.

Example: Apply an abstract algorithm to a specific example/situation, i.e. execute statements w/ values.
Quote: “Then we insert a 9 into the list**.”**

3.2. Implementing – applying a procedure to an unfamiliar task (e.g., use Newton’s second law in situations in which it is appropriate)

Example: Write computer code to perform a specific procedure.
Quote: “x values colon, percent d”

4. **Analyze** –aid in fuller understanding or prelude to evaluation; used to convey the meaning or establish a conclusion of a communication, e.g. problem statement, design, solution, etc. **Note**: this includes syntax and logic.

4.1. Differentiating – distinguishing relevant from irrelevant parts or important from unimportant parts of presented material (e.g., Distinguish between relevant and irrelevant numbers in a mathematical word problem)

Example: Recognize which statements, groups of statements, or methods to check when an error occurs in an algorithm or program.

Quote: "Is there some kind of data structure that I should be using other than this, other than the linked list?"

4.2. Organizing – determining how elements fit or function within a structure (e.g., structure evidence in a historical description into evidence for and against a particular historical explanation)

Example: Determine the overall design pattern and structure of a computer statement, group of statements, or an entire program.

Quote: "The point was to keep the data structure, the data organized."

4.3. Attributing – determine a point of view, values, or intent underlying presented material (e.g., determine the point of view of the author of an essay in terms of his or her political perspective)

Example: Determine how the variables behave in a procedure.

Quote: "2 got added in now 2 is my minimum and then 3 got added so 3 is still greater than 2 so 2 is still my minimum."

5. **Evaluate** – make judgments based on criteria and standards. **Note**: this includes syntax and logic.

5.1. Checking – detecting inconsistencies or fallacies within a process or product; determining whether a process or product has internal consistency; detecting the effectiveness of a procedure as it is being implemented (e.g., determine if a scientist's conclusions follow from observed data)

Example: Indicating/Detecting the logical fallacies in conditional statements.

Quote: "I can check the variables in the loop"

5.2. Critiquing – detecting inconsistencies between a product and external criteria, determining whether a product has external consistency; detecting the appropriateness of a procedure for a given problem (e.g., judge which of two methods is the best way to solve a given problem)

Example: Recognize the consequences to using a specific procedure to answer a problem.

Quote: "leaving the j variable in will give us some more fun values"

6. **Create** – put elements together to form a coherent or functional whole; reorganize elements into a new pattern or structure

6.1. Generating – coming up with alternative hypotheses based on criteria (e.g., generate hypotheses to account for an observed phenomenon)

Example: Design an illustration of the entire data structure by putting together individual components.

Quote: "I'm going to re-draw an array list with some random data so I can get an idea of what the structure looks like."

6.2. Planning – devising a procedure for accomplishing some task (e.g., plan a research paper on a given historical topic). **Note**: this includes high-level planning statements about goals and sub-goals (often confused with 2.7).

Example: Create specifications for a computer program.
Quote: “I need to insert these in the right order, because it should be going to the front of the list not the back.”

6.3. Producing – inventing a product (e.g., build habits for a specific purpose)
Example: Develop a scheme for classifying the types of problems that use specific data structures.
Quote:

Knowledge

A. **Factual Knowledge** – the basic elements students must know to be acquainted with a discipline or solve problems in it.

   a. Knowledge of terminology – technical vocabulary, musical symbols
   Example: Define what a data structure is by giving its attributes and properties.
   Quote: “… you would have your basic array list with a head and a tail and when you want to push … you would just find the last element…”

   b. Knowledge of specific details and elements – major natural resources, reliable sources of information
   Example: Describe the difference in properties between data structures.
   Quote: “…you just have elements of each, an integers I guess and either an array or the nodes that you have in the list um**.”**

B. **Conceptual Knowledge** – the interrelationships among the basic elements within a larger structure that enable them to function together

   a. Knowledge of classifications and categories – periods of geological time, forms of business ownership
   Example: Recognize how a data structure applies to many problems or visa versa.
   Quote: “Ok so I was thinking either you have an array or a linked list of some sort…”

   b. Knowledge of principles and generalizations – Pythagorean theorem, law of supply and demand
   Example: Understanding of major principles for a specific data structure.
   Quote: “I just want to make it initialized by just go up by one, but, um”

   c. Knowledge of theories, models, and structures – theory of evolution, structure of congress
   Example: Describe the theory of an algorithm.
   Quote:

C. **Procedural Knowledge** – how to do something, methods of inquiry, and criteria for using skills, algorithms, techniques, and methods

   a. Knowledge of subject-specific skills and algorithms – skills used in painting with watercolors, whole-number division algorithm
   Example: Describe a specific procedure used to accomplish a goal or sub-goal.
   Quote: “so, ok, so the first initial in the list, i, in uh, ok, so the first thing in the array equals,”

   b. Knowledge of subject-specific techniques and methods – interviewing techniques, scientific method
   Example: Describe the operations needed for a specific data structure.
   Quote: “Uh, push you just increment the counter and then save the integer on to the list, or array, or whatever you want to call it.”

c. Knowledge of criteria for determining when to use appropriate procedures – criteria used to determine when to apply a procedure involving Newton's second law, criteria used to judge the feasibility of using a particular method to estimate business costs
Example: Define the elements used to judge correctness of a data structure.
Quote: "In O(1) time means you have to be able to jump right to it."

D. **Metacognitive Knowledge** – knowledge of cognition in general as well as awareness and knowledge of one's own cognition

a. Strategic knowledge – knowledge of outlining as a means of capturing the structure of a unit of subject matter in a textbook, knowledge of the use of heuristics.
Example: Re-reading the problem statement to plan the next goals and sub-goals.
Quote: "Alright, I am re-reading the problem to start off."

b. Knowledge about cognitive tasks, including appropriate contextual and conditional knowledge – knowledge of the types of tests particular teachers administer, knowledge of the cognitive demands of different tasks
Example: Describe/Make a reference to task difficulty for self.
Quote: "This will take longer if I create a for loop rather than using a while."

c. Self-knowledge – knowledge that critiquing essays is a personal strength, whereas writing essays is a personal weakness; awareness of one's own knowledge level
Example: Describe one's own ability to accomplish a task.
Quote: "I'm in a brain freeze figuring out how to do that."

Col J Systems/Software Development Methodology

Planning – To generate a high-level view of the intended project and determine the goals of the project.
Example: Stating the need to read the problem to determine what needs to be done.
Quote: "Alright, I am re-reading the problem to start off."

Requirements – the tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various users. These requirements include user and system requirements.
Example: Recognize the specific tasks required in creating a new program to solve a computational problem.
Quote: "If we do the square root then we probably need math."

Design -a process of problem-solving and planning for a software solution
Example: Write pseudo-code for a particular solution to a problem before writing the computer code.
Quote:

Implementation/Code - the process of writing and debugging/troubleshooting the source code written in a programming language.
Example: Describe the computer statements needed to write a program to solve a specific problem.

Quote: “make a z because we are going to need that too”

Verification/Testing - an empirical investigation conducted to provide users with information about the quality of the product. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors.
Example: Make judgments on the software quality based on observations from executing a piece of software.
Quote: “good, ok, just what I wanted”

Maintenance - the modification of a software product to correct faults, to improve performance or other attributes.
Example: Determine what needs to be altered to correct errors in the software.
Quote: “I need to initialize the variable in the for loop to get better results.”

Appendix J Version 2 of The Coding Document

Coding Definitions for Test of Computational Thinking research study   As of: 2/04/2009

Accepted abbreviations:

E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action

Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page

URL of current web page

Title of current book

putty.exe = code this when user is interacting with the ssh terminal

problem page = code this when user is interacting with the piece of paper with question to answer

design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions can code > 1 on one line

compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]

readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)

bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)

designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions

login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command

- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action

copy = Copy text; describe the text being copied
cut = Cut text; describe the text being cut
paste = Paste text
move = Moving the cursor around in the text editor; specify up, down, left, right and how many times.
whitesp = Creating newlines/white space in the program for readability
delete = Delete text by backspacing; describe the text that is being deleted
typing = Typing text in the editor; (The text is captured in column E)
exit = Exit the text editor;
save = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action
brwwith = Browsing within a content site; clicking a link on a content site that leads to another page in content site
brwbetw = Browsing between content sites = clicking a link on a content site that leads to a new content site.
goseng = Going to a search engine from the problem start page
sengsearch = Doing a search engine search
sengresult = Clicking on a search engine result.
sengresultnext = Browsing to the next (or previous) page of search engine results.
sitesearch = Doing a site search (ie, a within-site search engine search)
typurl = Typing a URL into the browser address bar.
back = Clicking BACK button
mouseover = Mousing over popup and cascading menus or over link text or headers.
copy = Copying info from a web site
scan = Scanning a page
scroll = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;
may do a small bit of paraphrasing as long as you keep all the original meaning;
when in doubt, transcribe every word.
eg, when P is reading aloud from the problem page, do **NOT** code **~~(reading prob page)~~**,
do code the words P is saying Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our

segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don't know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel? Is a sentinal vaue just like null or what? [E "You can use null as an ending, umhum, it is a terminating value"] ok	consider prob (what is the sentinel value) using external info: E consider prob (confusion re sentinel value)

Good segmenting: because you DO know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel?	consider prob (what is the sentinel value)
Excel line 2:		Is a sentinal vaue just like null or what?	consider prop (confusion re sentinel value)
Excel line 3:		[E "You can use null as an ending, umhum, it is a terminating value"] ok	using external info: E

Integrate info from other sources (eg experimenter's notes, experimenter's words) when needed to understand participant's verbal comments. If P says "This sentence is confusing" and the notes say "P pointed to 2nd sentence of problem page", include this comment in the transcript, in brackets.

Formatting

Put participant's words in without brackets.

Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**

If P is reading from the problem text or a website put in quotes, eg, **“**Let n represent the actual number of data values**.”**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, This sentence is confusing [P pointed to 3rd sentence of page]

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G Domain-Specific Schemata (see problem statement w/ goals and subtasks)

Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action; **OR**
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**; **OR**

- P finishes answering a particular goal or subtask, this is coded in red. **Eg 1**, completed goal 1, sub 1

These goals will mostly refer to user statement about the problem (see the problem statement), so they will refer to arrays, loops, products, summing, square root, and other concepts from the problem. See examples below.

Eg of: Goals explicitly mentioned by user

Eg 2: user comment="Now I am going to compute and print the square root of the sum of z."
Code="answer question; goal 2, subtask 1"

Eg of: Goals not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 3: user comment = "If we do the square root then we probably need math." Then the user types
"#include <math.h>"
From the user's words and keystrokes, you can **infer** a goal and subtask.
Code="answer question; goal 5, subtask 1"

Note: Include any specific details about the goal or answer, as well as any observations made by E, in parenthesis beside the goal. **Eg 4**, answer question; goal 3, subtask 1 (chooses a while loop) or completed goal 3, sub 1 (no sentinel value)

Col H Metacognitive Strategies can code > 1 on one line

Background: code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem.

read design code when P reads information from the piece of paper containing his/her design, drawings, and/or notes; also note what was read from the design.

read prob code when P reads or re-reads the problem statement or part of it; also note what part of problem read, **eg**, **read prob (1st sentence)**

consider prob code when P asks a question or makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing**)
eg, "Sentinel, what is a sentinel?", could code: **consider prob (confusion re sentinel value)**

task difficulty code whenever P mentions the difficulty of some part of their task. However, if P mentions the difficulty of part of the problem/question, code this under **consider prob**,
eg, "this is the same thing I had trouble with a second ago", code as **task difficulty**

time planning code whenever P discusses scheduling of activities or how he/she will allocate his/her time and effort.
eg, "First, I am going to work on initializing the arrays, and then I will print them out ", code as **time planning**

revisit goal code whenever P **leaves** a goal or sub-goal in working memory and then **returns** to the goal/sub-goal – also code what he/she is recycling in parenthesis, **eg**, **revisit goal (Goal 2, Subtask 1)**

knowledge elaboration code whenever P elaborates on the information read, seen, or heard – also code the knowledge that is being elaborated in parenthesis,
eg, "ok, so, i is zero, put in numbers because that helps", code as **knowledge elaboration (evaluation strategy for answer)**

interest statement code whenever P mentions some part of their task is interesting – also indicate whether the statement was positive or negative, as well as the task of interest.
eg, "no, for loop again, god", code as **interest statement (negative for loops)**

Col I Programming Strategies Schemata can code > 1 on one line
Background: code whenever P makes a statement about his/her answer or makes a statement used in the development of an answer, which may be a program or a verbal solution to a computational problem.

using external info code whenever P uses an external source of information to help with understanding the problem or answering the problem. In addition, code where the information is coming from, i.e. book, internet, old program, or E. Note: if P is doing any of the following while using an external source of information, code in parenthesis:

paraphrasing code whenever P **summarizes** the gist of some information coming from E, the web, a book, or an old program. Note: if P summarizes part of the problem or question, code this under **consider prob**

support code when P mentions that 2+ pieces of knowledge **support** or **agree with** or **confirm** or **provide evidence for** each other– also code the 2 pieces of knowledge. Note: this is a positive content evaluation.
eg, P retrieves a potential piece of code from own knowledge but says she needs to verify it; P searches and finds code in a book; then P enters this answer (or accepts previously entered answer as ok). The verifying info in the book would be coded as **support**-ing the potential answer from own knowledge, **using external info: book (support: book verify own)**

lack of support code when P mentions a **contradiction** or **disagreement** or **lack of support** between 2 or more pieces of knowledge – also code the 2 pieces of knowledge. Note: this is a negative evaluation

site credibility code whenever P refers to the **credibility** or **believability** or **quality** or **accuracy** of the current external source of info, or mentions **high or low confidence or trust** in the info.

requirements code when P makes a statement about or recognizes needs and conditions to meet for specific tasks in the solution, taking account of the possibly conflicting requirements. These requirements include any (user and system requirements) used to determine solution; also note what was required.

eg, “If we do the square root then we probably need math.”, code as **requirements (math.h)**

design code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.
eg, P writes pseudo-code for creating arrays in the solution to a problem before writing the computer code, code as **design (create arrays pseudo-code)**

write code code when P writes code or mentions wanting to write code for the solution,
eg, **write code:** - also code the details

using prior knowl code whenever P searches memory for relevant prior knowledge or makes statements that use his/her own prior knowledge either before performing a task or during task performance, Note: this includes general statements about the code, self-questioning/explaining statements about one’s answer, goals, sub-goals, code, etc., and low-level planning statements about operations that are possible, postponed or intended – also code the low-level schemata.
eg 1, “I just want to make it initialized by just go up by one, but, um”, could code as **using prior knowl (initialize array to contiguous numbers using loop)**
eg 2, “z sub i equals x sub i plus y sub i”, could code as **using prior knowl (store product of 2 arrays in 3rd array)**
eg 3, “We can either use a for loop or a while loop to initialize x and y”, could code as **using prior knowl (plan loop type)**

need to learn code when P explicitly states needing to learn or find out more about a particular topic **OR** when P asks a question & uses an external source of information to answer, i.e. help-seeking behavior – also code the topic in parentheses if unclear. Note: If P asks a question regarding the problem statement, code this under **consider prob**.
eg, “what is the syntax for filling up an array?”, code as **need to learn (array initialization syntax)**.

recycle goal code whenever P **reuses** a goal or sub-goal in working memory. This includes cutting and pasting code from one goal to answer another goal – also code what he/she is recycling in parenthesis.

confidence code whenever P makes a high or low confidence statement about his/her code or answer. These include judgments of knowing and feeling of knowing statements – also code details in parenthesis.
eg, “I know I should use a for loop, but I don’t like for loops”, **confidence (low re for loop)**

debug code whenever P mentions testing the solution, considers his/her answer, or performs the act of testing/validating the solution. **eg**, **debug:** - also code the details

incremental testing code whenever P quickly tests or mentions the need to test small pieces of code as they are added (by any of the methods below) Note: This may include commenting out code to see what works

inspection code whenever P checks code via visual inspection and compares it to own knowledge or knowledge from an external source. This includes when P

- reads computer code he/she has written; also note what was read from the program **OR**
- is evaluating or considering whether some information (eg, a partial answer) is a **complete or correct answer**

eg, P is scanning over his/her loop and evaluating whether the loop is correct or complete; could code **inspection (evaluating re loop)**

compile code whenever P tests code via compiling it and inspecting the compiler error messages

execute using output code whenever P tests by executing a compiled program and comparing the actual and expected output

execute using internal data code whenever P tests by executing a compiled program and using test-write-statements or a debugging application/environment to inspect internal program data

diagnose code whenever P uses reasoning strategies to identify the cause of incorrect output or internal data (eg, backtracking from output code that gave bad output to prior code that provided data to the output code)

hypothesize code when P poses possible questions and answers about the error he/she is diagnosing – also code the details of the generated hypothesis(es) in parenthesis

eg, “It might be the type of brackets I used in the array declaration causing the error”, could code as **diagnose: hypothesize (brackets in array creation)**

fix code code whenever P changes code after using some of the above debugging methods

Appendix K Version 3 of The Coding Document

Coding Definitions for Test of Computational Thinking research study As of: 2/19/2009

Accepted abbreviations:
E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action
Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page
URL of current web page
Title of current book
putty.exe = code this when user is interacting with the ssh terminal
problem page = code this when user is interacting with the piece of paper with question to answer
design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions can code > 1 on one line
compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]
readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)
bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)
designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions
login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command
- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action

copy = Copy text; describe the text being copied
cut = Cut text; describe the text being cut
paste = Paste text
move = Moving the cursor around in the text editor; specify up, down, left, right and how many times.
whitesp = Creating newlines/white space in the program for readability
delete = Delete text by backspacing; describe the text that is being deleted
typing = Typing text in the editor; (The text is captured in column E)
exit = Exit the text editor;
save = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action
brwwith = Browsing within a content site; clicking a link on a content site that leads to another page in content site
brwbetw = Browsing between content sites = clicking a link on a content site that leads to a new content site.
goseng = Going to a search engine from the problem start page
sengsearch = Doing a search engine search
sengresult = Clicking on a search engine result.
sengresultnext = Browsing to the next (or previous) page of search engine results.
sitesearch = Doing a site search (ie, a within-site search engine search)
typurl = Typing a URL into the browser address bar.
back = Clicking BACK button
mouseover = Mousing over popup and cascading menus or over link text or headers.
copy = Copying info from a web site
scan = Scanning a page
scroll = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;
may do a small bit of paraphrasing as long as you keep all the original meaning;
when in doubt, transcribe every word.
eg, when P is reading aloud from the problem page, do **NOT** code ~~**(reading prob page)**~~, **do code the words P is saying** Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our

segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don't know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel? Is a sentinal vaue just like null or what? [E "You can use null as an ending, umhum, it is a terminating value"] ok	consider prob (what is the sentinel value) using external info: E consider prob (confusion re sentinel value)

Good segmenting: because you DO know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel?	consider prob (what is the sentinel value)
Excel line 2:		Is a sentinal vaue just like null or what?	consider prop (confusion re sentinel value)
Excel line 3:		[E "You can use null as an ending, umhum, it is a terminating value"] ok	using external info: E

Integrate info from other sources (eg experimenter's notes, experimenter's words) when needed to understand participant's verbal comments. If P says "This sentence is confusing" and the notes say "P pointed to 2nd sentence of problem page", include this comment in the transcript, in brackets.

Formatting

Put participant's words in without brackets.

Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**

If P is reading from the problem text or a website put in quotes, eg, **"**Let n represent the actual number of data values.**"**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, This sentence is confusing [P pointed to 3rd sentence of page]

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G Goals (see problem statement w/ goals and subtasks)

Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action; **OR**
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**; **OR**

- P finishes answering a particular goal or subtask, this is coded in red. **Eg 1**, completed goal 1, sub 1

These goals will mostly refer to user statement about the problem (see the problem statement), so they will refer to arrays, loops, products, summing, square root, and other concepts from the problem. See examples below.

Eg of: Goals explicitly mentioned by user

Eg 2: user comment="Now I am going to compute and print the square root of the sum of z."
Code="answer question; goal 2, subtask 1"

Eg of: Goals not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 3: user comment = "If we do the square root then we probably need math." Then the user types
"#include <math.h>"
From the user's words and keystrokes, you can **infer** a goal and subtask.
Code="answer question; goal 5, subtask 1"

Note: Include any specific details about the goal or answer, as well as any observations made by E, in parenthesis beside the goal. **Eg 4**, answer question; goal 3, subtask 1 (chooses a while loop) or completed goal 3, sub 1 (no sentinel value)

Col H Metacognitive Strategies can code > 1 on one line

Background: code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem.

read design code when P reads information from the piece of paper containing his/her design, drawings, and/or notes; also note what was read from the design.

read prob code when P reads or re-reads the problem statement or part of it; also note what part of problem read, **eg**, **read prob (1st sentence)**

consider prob code when P asks a question or makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing** or questions about the **next task in the problem**)
eg, "Sentinel, what is a sentinel?", could code: **consider prob (confusion re sentinel value)**
eg, "and now I need to print them out, right. Let's see, where is it?", could code: **consider prob (determine if next goal is to print out arrays)**

task difficulty code whenever P mentions the difficulty of some part of their task. However, if P mentions the difficulty of part of the problem/question, code this under **consider prob**,
eg, "this is the same thing I had trouble with a second ago", code as **task difficulty**

time planning code whenever P discusses scheduling of activities or how he/she will allocate his/her time and effort.
eg, "First, I am going to work on initializing the arrays, and then I will print them out ", code as **time planning**

revisit goal code whenever P **leaves** a goal or sub-goal in working memory and then **returns** to the goal/sub-goal – also code what he/she is revisiting in parenthesis, **eg**, **revisit goal (Goal 2, Subtask 1)**

knowledge elaboration code whenever P elaborates on the information read, seen, or heard – also code the knowledge that is being elaborated in parenthesis,
eg, “ok, so, i is zero, put in numbers because that helps”, code as **knowledge elaboration (evaluation strategy for answer)**

interest statement code whenever P mentions some part of their task is interesting – also indicate whether the statement was positive or negative, as well as the task of interest.
eg, “no, for loop again, god”, code as **interest statement (negative for loops)**

Col I Programming Strategies Schemata can code > 1 on one line

Background: code whenever P makes a statement about his/her answer or makes a statement used in the development of an answer, which may be a program or a verbal solution to a computational problem.

using external info code whenever P uses an external source of information to help with understanding the problem or answering the problem. In addition, code where the information is coming from, i.e. book, internet, old program, or E. Note: if P is doing any of the following while using an external source of information, code in parenthesis:

paraphrasing code whenever P **summarizes** the gist of some information coming from E, the web, a book, or an old program. Note: if P summarizes part of the problem or question, code this under **consider prob**

support code when P mentions that 2+ pieces of knowledge **support** or **agree with** or **confirm** or **provide evidence for** each other– also code the 2 pieces of knowledge. Note: this is a positive content evaluation.
eg, P retrieves a potential piece of code from own knowledge but says she needs to verify it; P searches and finds code in a book; then P enters this answer (or accepts previously entered answer as ok). The verifying info in the book would be coded as **support**-ing the potential answer from own knowledge, **using external info: book (support: book verify own)**

lack of support code when P mentions a **contradiction** or **disagreement** or **lack of support** between 2 or more pieces of knowledge – also code the 2 pieces of knowledge. Note: this is a negative evaluation

site credibility code whenever P refers to the **credibility** or **believability** or **quality** or **accuracy** of the current external source of info, or mentions **high or low confidence or trust** in the info.

requirements code when P makes a statement about or recognizes needs and conditions to meet for specific tasks in the solution, taking account of the possibly conflicting requirements. These requirements include any (user and system requirements) used to determine solution; also note what was required.

eg, "If we do the square root then we probably need math.", code as **requirements (math.h)**

design code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.

eg, P writes pseudo-code for creating arrays in the solution to a problem before writing the computer code, code as **design (create arrays pseudo-code)**

write code code when P writes code or mentions wanting to write code for the solution,

eg, **write code:** - also code the details

using prior knowl code whenever P searches memory for relevant prior knowledge or makes statements that use his/her own prior knowledge either before performing a task or during task performance, Note: this includes general statements about the code, considering options for answer, self-questioning/explaining statements about one's answer, goals, sub-goals, code, etc., and low-level planning statements about operations that are possible, postponed or intended – also code the low-level schemata.

eg 1, "I just want to make it initialized by just go up by one, but, um", could code as **using prior knowl (initialize array to contiguous numbers using loop)**

eg 2, "z sub i equals x sub i plus y sub i", could code as **using prior knowl (store product of 2 arrays in 3rd array)**

eg 3, "We can either use a for loop or a while loop to initialize x and y", could code as **using prior knowl (plan loop type)**

need to learn code when P explicitly states needing to learn or find out more about a particular topic **OR** when P asks a question & uses an external source of information to answer, i.e. help-seeking behavior – also code the topic in parentheses if unclear. Note: If P asks a question regarding the problem statement, code this under **consider prob**.

eg, "what is the syntax for filling up an array?", code as **need to learn (array initialization syntax)**.

recycle goal code whenever P reuses a goal or sub-goal in working memory. This includes cutting and pasting code from one goal to answer another goal – also code what he/she is recycling in parenthesis.

confidence code whenever P makes a high or low confidence statement about his/her code or answer. These include judgments of knowing and feeling of knowing statements – also code details in parenthesis.

eg, "I know I should use a for loop, but I don't like for loops", **confidence (low re for loop)**

learn syntax by guess/compile code whenever P makes statements about guessing the computer syntax or mentions intentions of using the compiler to determine correctness.

eg, "probably easier to just test it out and see if, and see what works", code as **learn syntax by guess/compile**

debug code whenever P considers his/her answer or performs the act of testing/validating the solution.

eg, **debug:** - also code the details

incremental testing code whenever P quickly tests or mentions the need to test small pieces of code as they are added (by any of the methods below) Note: This may include commenting out code to see what works

inspection code whenever P checks code via visual inspection and compares it to own knowledge or knowledge from an external source. This includes when P

- reads computer code he/she has written; also note what was read from the program **OR**
- is evaluating or considering whether some information (eg, a partial answer) is a **complete or correct answer**

eg, P is scanning over his/her loop and evaluating whether the loop is correct or complete; could code **inspection (evaluating re loop)**

compile code whenever P tests code via compiling it and inspecting the compiler error messages

execute using output code whenever P tests by executing a compiled program and comparing the actual and expected output

execute using internal data code whenever P tests by executing a compiled program and using test-write-statements or a debugging application/environment to inspect internal program data

diagnose code whenever P uses reasoning strategies to identify the cause of incorrect output or internal data (eg, backtracking from output code that gave bad output to prior code that provided data to the output code) **OR** uses statements identifying the cause of an error without explicitly stating the reasoning.

hypothesize code when P poses possible questions and answers about the error he/she is diagnosing – also code the details of the generated hypothesis(es) in parenthesis

eg, "It might be the type of brackets I used in the array declaration causing the error", could code as **diagnose: hypothesize (brackets in array creation)**

fix code code whenever P changes code after debugging using some of the above methods

Appendix L Version 4 of The Coding Document

Coding Definitions for Test of Computational Thinking research study As of: 3/10/2009

Accepted abbreviations:
E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action
Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page
URL of current web page
Title of current book
putty.exe = code this when user is interacting with the ssh terminal
problem page = code this when user is interacting with the piece of paper with question to answer
design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions can code > 1 on one line
compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]
readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)
bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)
designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions
login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command
- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action

copy = Copy text; describe the text being copied

cut = Cut text; describe the text being cut

paste = Paste text

move = Moving the cursor around in the text editor; specify up, down, left, right and how many times.

whitesp = Creating newlines/white space in the program for readability

delete = Delete text by backspacing; describe the text that is being deleted

typing = Typing text in the editor; (The text is captured in column E)

exit = Exit the text editor;

save = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action

brwwith = Browsing within a content site; clicking a link on a content site that leads to another page in content site

brwbetw = Browsing between content sites = clicking a link on a content site that leads to a new content site.

goseng = Going to a search engine from the problem start page

sengsearch = Doing a search engine search

sengresult = Clicking on a search engine result.

sengresultnext = Browsing to the next (or previous) page of search engine results.

sitesearch = Doing a site search (ie, a within-site search engine search)

typurl = Typing a URL into the browser address bar.

back = Clicking BACK button

mouseover = Mousing over popup and cascading menus or over link text or headers.

copy = Copying info from a web site

scan = Scanning a page

scroll = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;
may do a small bit of paraphrasing as long as you keep all the original meaning;
when in doubt, transcribe every word.
eg, when P is reading aloud from the problem page, do **NOT** code ~~**(reading prob page)**~~,
do code the words P is saying Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our

segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don't know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel? Is a sentinal vaue just like null or what? [E "You can use null as an ending, umhum, it is a terminating value"] ok	consider prob (what is the sentinel value) using external info: E consider prob (confusion re sentinel value)

Good segmenting: because you DO know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel?	consider prob (what is the sentinel value)
Excel line 2:		Is a sentinal vaue just like null or what?	consider prop (confusion re sentinel value)
Excel line 3:		[E "You can use null as an ending, umhum, it is a terminating value"] ok	using external info: E

Integrate info from other sources (eg experimenter's notes, experimenter's words) when needed to understand participant's verbal comments. If P says "This sentence is confusing" and the notes say "P pointed to 2nd sentence of problem page", include this comment in the transcript, in brackets.

Formatting

Put participant's words in without brackets.

Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**

If P is reading from the problem text or a website put in quotes, eg, **"**Let n represent the actual number of data values**."**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, This sentence is confusing [P pointed to 3rd sentence of page]

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G Goals (see problem statement w/ goals and subtasks)

Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action; **OR**
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**; **OR**

- P finishes answering a particular goal or subtask, this is coded in red. **Eg 1**, completed goal 1, sub 1

These goals will mostly refer to user statement about the problem (see the problem statement), so they will refer to arrays, loops, products, summing, square root, and other concepts from the problem. See examples below.

Eg of: Goals explicitly mentioned by user

Eg 2: user comment="Now I am going to compute and print the square root of the sum of z."
Code="answer question; goal 2, subtask 1"

Eg of: Goals not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 3: user comment = "If we do the square root then we probably need math." Then the user types
"#include <math.h>"
From the user's words and keystrokes, you can **infer** a goal and subtask.
Code="answer question; goal 5, subtask 1"

Note: Include any specific details about the goal or answer, as well as any observations made by E, in parenthesis beside the goal. **Eg 4**, answer question; goal 3, subtask 1 (chooses a while loop) or completed goal 3, sub 1 (no sentinel value)

Col H Metacognitive Strategies can code > 1 on one line

Background: code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem.

read design code when P reads information from the piece of paper containing his/her design, drawings, and/or notes; also note what was read from the design.

read prob code when P reads or re-reads the problem statement or part of it; also note what part of problem read, **eg**, **read prob (1st sentence)**

consider prob code when P asks a question or makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing** or questions about the **next task in the problem**)
eg, "Sentinel, what is a sentinel?", could code: **consider prob (confusion re sentinel value)**
eg, "and now I need to print them out, right. Let's see, where is it?", could code: **consider prob (determine if next goal is to print out arrays)**

task difficulty code whenever P mentions the difficulty of some part of their task. However, if P mentions the difficulty of part of the problem/question, code this under **consider prob**,
eg, "this is the same thing I had trouble with a second ago", code as **task difficulty**

time planning code whenever P discusses scheduling of activities or how he/she will allocate his/her time and effort.
eg, "First, I am going to work on initializing the arrays, and then I will print them out ", code as **time planning**

revisit goal code whenever P **leaves** a goal or sub-goal in working memory and then **returns** to the goal/sub-goal – also code what he/she is revisiting in parenthesis, **eg**, **revisit goal (Goal 2, Subtask 1)**

knowledge elaboration code whenever P elaborates on the information read, seen, or heard – also code the knowledge that is being elaborated in parenthesis,

eg, “ok, so, i is zero, put in numbers because that helps”, code as **knowledge elaboration (evaluation strategy for answer)**

interest statement code whenever P mentions some part of their task is interesting – also indicate whether the statement was positive or negative, as well as the task of interest.

eg, “no, for loop again, god”, code as **interest statement (negative for loops)**

mentally inspect/exe code whenever P inspects a mental representation of code rather than actual written code or executes code mentally to evaluate correctness.

eg, “that equals one, then that will go to the second thing in the list equals one, so everything in the list will equal one and then,”, code as **mentally inspect/exe (predicting effects of code)**

Col I Programming Strategies Schemata can code > 1 on one line

Background: code whenever P makes a statement about his/her answer or makes a statement used in the development of an answer, which may be a program or a verbal solution to a computational problem.

using external info code whenever P uses an external source of information to help with understanding the problem or answering the problem. In addition, code where the information is coming from, i.e. book, internet, old program, or E. Note: if P is doing any of the following while using an external source of information, code in parenthesis:

paraphrasing code whenever P **summarizes** the gist of some information coming from E, the web, a book, or an old program. Note: if P summarizes part of the problem or question, code this under **consider prob**

support code when P mentions that 2+ pieces of knowledge **support** or **agree with** or **confirm** or **provide evidence for** each other– also code the 2 pieces of knowledge. Note: this is a positive content evaluation.

eg, P retrieves a potential piece of code from own knowledge but says she needs to verify it; P searches and finds code in a book; then P enters this answer (or accepts previously entered answer as ok). The verifying info in the book would be coded as **support**-ing the potential answer from own knowledge, **using external info: book (support: book verify own)**

lack of support code when P mentions a **contradiction** or **disagreement** or **lack of support** between 2 or more pieces of knowledge – also code the 2 pieces of knowledge. Note: this is a negative evaluation

site credibility code whenever P refers to the **credibility** or **believability** or **quality** or **accuracy** of the current external source of info, or mentions **high or low confidence or trust** in the info.

requirements code when P makes a statement about or recognizes needs and conditions to meet for specific tasks in the solution, taking account of the possibly conflicting requirements. These requirements include any (user and system requirements) used to determine solution; also note what was required.

eg, "If we do the square root then we probably need math.", code as **requirements (math.h)**

design code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.

eg, P writes pseudo-code for creating arrays in the solution to a problem before writing the computer code, code as **design (create arrays pseudo-code)**

write code code when P writes code or mentions wanting to write code for the solution,

eg, **write code:** - also code the details

using prior knowl code whenever P searches memory for relevant prior knowledge or makes statements that use his/her own prior knowledge either before performing a task or during task performance, Note: this includes general statements about the code, considering options for answer, self-questioning/explaining statements about one's answer, goals, sub-goals, code, etc., and low-level planning statements about operations that are possible, postponed or intended – also code the low-level schemata.

eg 1, "I just want to make it initialized by just go up by one, but, um", could code as **using prior knowl (initialize array to contiguous numbers using loop)**

eg 2, "z sub i equals x sub i plus y sub i", could code as **using prior knowl (store product of 2 arrays in 3rd array)**

eg 3, "We can either use a for loop or a while loop to initialize x and y", could code as **using prior knowl (plan loop type)**

need to learn code when P explicitly states needing to learn or find out more about a particular topic **OR** when P asks a question & uses an external source of information to answer, i.e. help-seeking behavior, **OR** when P follows a question with a learn syntax by guess/compile,– also code the topic in parentheses if unclear. Note: If P asks a question regarding the problem statement, code this under **consider prob**.

eg, "what is the syntax for filling up an array?", code as **need to learn (array initialization syntax)**.

eg, "how do I add the elements inside the loop?", code as **need to learn (array initialization syntax)**.

recycle goal code whenever P reuses a goal or sub-goal in working memory. This includes cutting, pasting, and editing code from one goal to answer another goal – also code what he/she is recycling in parenthesis.

confidence code whenever P makes a high, medium, or low confidence statement about his/her code or answer. These can include judgment of knowing and feeling of knowing statements – also code details in parenthesis.

eg, "I know I should use a for loop, but I don't like for loops", **confidence (low re for loop)**

eg, “I think I can declare and initialize the i inside the loop ”, **confidence (medium re knowl about i in loop)**

learn syntax by guess/compile code whenever P makes statements about guessing the computer syntax or mentions intentions of using the compiler to determine correctness.

eg, “probably easier to just test it out and see if, and see what works”, code as **learn syntax by guess/compile**

debug code whenever P considers his/her answer or performs the act of testing/validating the solution.

eg, **debug:** - also code the details

incremental testing code whenever P quickly tests or mentions the need to test small pieces of code as they are added (by any of the methods below) Note: This may include commenting out code to see what works

inspection code whenever P checks code via visual inspection and compares it to own knowledge or knowledge from an external source. This includes when P

- reads computer code he/she has written; also note what was read from the program **OR**
- is evaluating or considering whether some information (eg, a partial answer) is a **complete or correct answer**

eg, P is scanning over his/her loop and evaluating whether the loop is correct or complete; could code **inspection (evaluating re loop)**

compile code whenever P tests code via compiling it and inspecting the compiler error messages

execute using output code whenever P tests or mentions testing by executing a compiled program and comparing the actual and expected output

execute using internal data code whenever P tests or mentions testing by executing a compiled program and using test-write-statements or a debugging application/environment to inspect internal program data

diagnose code whenever P uses reasoning strategies to identify the cause of incorrect output or internal data (eg, backtracking from output code that gave bad output to prior code that provided data to the output code) **OR** uses statements identifying the cause of an error without explicitly stating the reasoning.

hypothesize code when P poses possible questions and answers about the error he/she is diagnosing – also code the details of the generated hypothesis(es) in parenthesis

eg, “It might be the type of brackets I used in the array declaration causing the error”, could code as **diagnose: hypothesize (brackets in array creation)**

fix code code whenever P changes code after debugging using some of the above methods

Appendix M Version 5 of The Coding Document

Coding Definitions for Test of Computational Thinking research study **As of: 4/08/2009**

Accepted abbreviations:
E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action
Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page
URL of current web page
Title of current book
putty.exe = code this when user is interacting with the ssh terminal
problem page = code this when user is interacting with the piece of paper with question to answer
design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions can code > 1 on one line
compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]
readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)
bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)
designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions
login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command
- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action

copy = Copy text; describe the text being copied

cut = Cut text; describe the text being cut

paste = Paste text

move = Moving the cursor around in the text editor; specify up, down, left, right and how many times.

whitesp = Creating newlines/white space in the program for readability

delete = Delete text by backspacing; describe the text that is being deleted

typing = Typing text in the editor; (The text is captured in column E)

exit = Exit the text editor;

save = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action

brwwith = Browsing within a content site; clicking a link on a content site that leads to another page in content site

brwbetw = Browsing between content sites = clicking a link on a content site that leads to a new content site.

goseng = Going to a search engine from the problem start page

sengsearch = Doing a search engine search

sengresult = Clicking on a search engine result.

sengresultnext = Browsing to the next (or previous) page of search engine results.

sitesearch = Doing a site search (ie, a within-site search engine search)

typurl = Typing a URL into the browser address bar.

back = Clicking BACK button

mouseover = Mousing over popup and cascading menus or over link text or headers.

copy = Copying info from a web site

scan = Scanning a page

scroll = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;

may do a small bit of paraphrasing as long as you keep all the original meaning;

when in doubt, transcribe every word.

eg, when P is reading aloud from the problem page, do **NOT** code ~~**(reading prob page)**~~,

do code the words P is saying Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our

segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don't know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel? Is a sentinal vaue just like null or what? [E "You can use null as an ending, umhum, it is a terminating value"] ok	consider prob (what is the sentinel value) using external info: E consider prob (confusion re sentinel value)

Good segmenting: because you DO know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel?	consider prob (what is the sentinel value)
Excel line 2:		Is a sentinal vaue just like null or what?	consider prop (confusion re sentinel value)
Excel line 3:		[E "You can use null as an ending, umhum, it is a terminating value"] ok	using external info: E

Integrate info from other sources (eg experimenter's notes, experimenter's words) when needed to understand participant's verbal comments. If P says "This sentence is confusing" and the notes say "P pointed to 2nd sentence of problem page", include this comment in the transcript, in brackets.

Formatting

Put participant's words in without brackets.

Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**

If P is reading from the problem text or a website put in quotes, eg, **"**Let n represent the actual number of data values**."**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, **This sentence is confusing [P pointed to 3rd sentence of page]**

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G Goals (see problem statement w/ goals and subtasks)

Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action; **OR**
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**; **OR**

- P finishes answering a particular goal or subtask, this is coded in red. **Eg 1**, completed goal 1, sub 1

These goals will mostly refer to user statement about the problem (see the problem statement), so they will refer to arrays, loops, products, summing, square root, and other concepts from the problem. See examples below.

Eg of: Goals explicitly mentioned by user

Eg 2: user comment="Now I am going to compute and print the square root of the sum of z."
Code="answer question; goal 2, subtask 1"

Eg of: Goals not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 3: user comment = "If we do the square root then we probably need math." Then the user types
"#include <math.h>"
From the user's words and keystrokes, you can **infer** a goal and subtask.
Code="answer question; goal 5, subtask 1"

Note: Include any specific details about the goal or answer, as well as any observations made by E, in parenthesis beside the goal. **Eg 4**, answer question; goal 3, subtask 1 (chooses a while loop) or completed goal 3, sub 1 (no sentinel value)

Col H Metacognitive Strategies can code > 1 on one line

Background: code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem.

read design code when P reads information from the piece of paper containing his/her design, drawings, and/or notes; also note what was read from the design.

consider design code when P asks a question or makes a statement about his/her design
eg, "", could code: **consider design**

read compiler code when P reads information from the compiler; also note what was read from the compiler.

consider compiler code when P asks a question or makes a statement about the compiler message **eg**, "uh, for loop [referring to the error message given by the compiler]", could code: **consider compiler (evaluate for loop as error)**

read output code when P reads information from the output after executing a program; also note what was read.

consider output code when P asks a question or makes a statement about the program output
eg, "good, ok, just what I wanted [output from program x is printed with the values 1 - 10]", could code: **consider output (output matched expected)**

read prob code when P reads or re-reads the problem statement or part of it; also note what part of problem read, **eg**, **read prob (1st sentence)**

consider prob code when P asks a question or makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing** or questions about the **next task in the problem**)

eg, “Sentinel, what is a sentinel?”, could code: **consider prob (confusion re sentinel value)**

eg, “and now I need to print them out, right. Let's see, where is it?”, could code: **consider prob (determine if next goal is to print out arrays)**

task difficulty code whenever P mentions the difficulty of some part of their task. However, if P mentions the difficulty of part of the problem/question, code this under **consider prob**,

eg, “this is the same thing I had trouble with a second ago”, code as **task difficulty**

[type] planning; [type = general|time|etc.] code whenever P discusses general scheduling of activities or how he/she will allocate his/her time and effort.

eg, “First, I am going to work on initializing the arrays, and then I will print them out.”, code as **time planning**

eg, “I will name them exactly what they are in the program, uh in the problem.”, code as **general planning**

revisit goal code whenever P **leaves** a goal or sub-goal in working memory and then **returns** to the goal/sub-goal – also code what he/she is revisiting in parenthesis, **eg**, **revisit goal (Goal 2, Subtask 1)**

knowledge elaboration code whenever P elaborates on the information read, seen, or heard – also code the knowledge that is being elaborated in parenthesis,

eg, “ok, so, i is zero, put in numbers because that helps”, code as **knowledge elaboration (evaluation strategy for answer)**

interest statement code whenever P mentions some part of their task is interesting – also indicate whether the statement was positive or negative, as well as the task of interest.

eg, “no, for loop again, god”, code as **interest statement (negative for loops)**

mentally inspect code whenever P inspects a mental representation of code rather than actual written code and/or executes non-existent code mentally to evaluate correctness.

eg, “that equals one, then that will go to the second thing in the list equals one, so everything in the list will equal one and then,”, code as **mentally inspect**

Col I Programming Strategies Schemata can code > 1 on one line

Background: code whenever P makes a statement about his/her answer or makes a statement used in the development of an answer, which may be a program or a verbal solution to a computational problem.

using external info code whenever P uses an external source of information to help with understanding the problem or answering the problem. In addition, code where the information is coming from, i.e. book, internet, old program, or E. Note: if P is doing any of the following while using an external source of information, code in parenthesis:

paraphrasing code whenever P **summarizes** the gist of some information coming from E, the web, a book, or an old program. Note: if P summarizes part of the problem or question, code this under **consider prob**

support code when P mentions that 2+ pieces of knowledge **support** or **agree with** or **confirm** or **provide evidence for** each other– also code the 2 pieces of knowledge. Note: this is a positive content evaluation.
eg, P retrieves a potential piece of code from own knowledge but says she needs to verify it; P searches and finds code in a book; then P enters this answer (or accepts previously entered answer as ok). The verifying info in the book would be coded as **support**-ing the potential answer from own knowledge, **using external info: book (support: book verify own)**

lack of support code when P mentions a **contradiction** or **disagreement** or **lack of support** between 2 or more pieces of knowledge – also code the 2 pieces of knowledge. Note: this is a negative evaluation

site credibility code whenever P refers to the **credibility** or **believability** or **quality** or **accuracy** of the current external source of info, or mentions **high or low confidence or trust** in the info.

requirements code when P makes a statement about or recognizes needs and conditions to meet for specific tasks in the solution, taking account of the possibly conflicting requirements. These requirements include any (user and system requirements) used to determine solution; also note what was required.
eg, "If we do the square root then we probably need math.", code as **requirements (math.h)**

design code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.
eg, P writes pseudo-code for creating arrays in the solution to a problem before writing the computer code, code as **design (create arrays pseudo-code)**

write code code when P writes code or mentions wanting to write code for the solution,
eg, **write code:** - also code the details

using prior knowl code whenever P searches memory for relevant prior knowledge or makes statements that use his/her own prior knowledge either before performing a task or during task performance, Note: this includes general statements about the code, considering options for answer, self-questioning/explaining statements about one's answer, goals, sub-goals, code, etc., and low-level planning statements about operations that are possible, postponed or intended – also code the low-level schemata.
eg 1, "I just want to make it initialized by just go up by one, but, um", could code as **using prior knowl (initialize array to contiguous numbers using loop)**
eg 2, "z sub i equals x sub i plus y sub i", could code as **using prior knowl (store product of 2 arrays in 3rd array)**
eg 3, "We can either use a for loop or a while loop to initialize x and y", could code as **using prior knowl (plan loop type)**

need to learn code when P explicitly states needing to learn or find out more about a particular topic **OR** when P asks a question & uses an external source of information to answer, i.e. help-seeking behavior, **OR** when P follows a question with a learn – also code the topic in parentheses if unclear. Note: If P asks a question regarding the problem statement, code this under **consider prob**.

eg, "what is the syntax for filling up an array?", code as **need to learn (array initialization syntax)**.

eg, "how do I add the elements inside the loop?", code as **need to learn (array initialization syntax)**.

recycle goal code whenever P reuses a goal or sub-goal in working memory. This includes cutting, pasting, and editing code from one goal to answer another goal – also code what he/she is recycling in parenthesis.

confidence code whenever P makes a high, medium, or low confidence statement about his/her code or answer. These can include judgment of knowing and feeling of knowing statements – also code details in parenthesis.

eg, "I know I should use a for loop, but I don't like for loops", **confidence (low re for loop)**

eg, "I think I can declare and initialize the i inside the loop ", **confidence (medium re knowl about i in loop)**

learn syntax by guess/compile code whenever P makes statements about guessing the computer syntax or mentions intentions of using the compiler to determine correctness.

eg, "probably easier to just test it out and see if, and see what works", code as **learn syntax by guess/compile**

debug code whenever P considers his/her answer or performs the act of testing/validating the solution.

eg, **debug:** - also code the details

incremental testing code whenever P quickly tests or mentions the need to test small pieces of code as they are added (by any of the methods below) Note: This may include commenting out code to see what works

inspection: [compareto]; [compareto = own|output|compiler|book|etc.] code whenever P checks code via visual inspection and compares it to own knowledge or knowledge from an external source such as the compiler, a book, or output from executing the program. This includes when P

- reads computer code he/she has written; also note what was read from the program **OR**
- is evaluating or considering whether some information (eg, a partial answer) is a **complete or correct answer**

eg, P is scanning over his/her loop and evaluating whether the loop is correct or complete; could code **inspection: own (evaluating re loop)**

eg, P is scanning over his/her loop and determining the possible error by comparing the code and output; could code **inspection: output (evaluating re loop)**

execute using knowl code whenever P uses his/her own knowledge to execute existing code mentally **eg**, "that equals one, then that will go to the second thing in the list equals one, so everything in the list will equal one and then,”, code as **execute using knowl (predicting effects of code)**

compile code whenever P tests code via compiling it and inspecting the compiler error messages

execute using output code whenever P tests or mentions testing by executing a compiled program and comparing the actual and expected output

execute using internal data code whenever P tests or mentions testing by executing a compiled program and using test-write-statements or a debugging application/environment to inspect internal program data

diagnose code whenever P uses reasoning strategies to identify the cause of incorrect output or internal data (eg, backtracking from output code that gave bad output to prior code that provided data to the output code) **OR** uses statements identifying the cause of an error without explicitly stating the reasoning.

hypothesize code when P poses possible questions and answers about the error he/she is diagnosing – also code the details of the generated hypothesis(es) in parenthesis

eg, “It might be the type of brackets I used in the array declaration causing the error”, could code as **diagnose: hypothesize (brackets in array creation)**

fix code code whenever P changes code after debugging using some of the above methods

Appendix N Version 6 of The Coding Document

Coding Definitions for Test of Computational Thinking research study **As of: 5/04/2009**

Accepted abbreviations:
E = experimenter; P = participant; MB = mumble; SH = sigh; HM = humming; TP = tapping;

Col A Time of each user computer action
Time from start of session; format = minutes:seconds (eg, 02:32.4 = 2 min 32.4 sec)

Col B Morae computer action (not used; may be hidden)

Col C Current user computer or paper page
URL of current web page
Title of current book
putty.exe = code this when user is interacting with the ssh terminal
problem page = code this when user is interacting with the piece of paper with question to answer
design page = code this when user is interacting with the blank piece of paper for writing down thoughts

Col D User computer actions can code > 1 on one line

compprob = computer problem; describe problem in column F (verbal transcript)
eg, [ssh terminal trouble]
readprob = Reading the problem document; describe what part is being read in column F(verbal transcript)
bookpage = Reading from a book; describe what is being indexed and read in column F(verbal transcript)
designpage = Writing on a sheet of paper; describe what is written in column F(verbal transcript)

SSH Terminal Actions
login = Logging onto a unix machine using the ssh terminal

unix_cmd = Entering a unix command; specify the command
- **list dir** = List the entries in the directory;
- **move file** = Move a file to another directory or to another file w/ different name;
- **copy file** = Copy a file to another directory or to another file w/ different name;
- **make dir** = Create a new directory;
- **change dir** = Change into a specific directory;
- **open file** = Opening a file using a text editor; describe the text editor and the file being opened
- **compile prog** = Compile a program; specify the compiler and the program being compiled
- **run prog** = Execute a program; specify the program being executed and describe the program output in column F (verbal transcript)

txt_ed = Working in a text editor; specify the action

- **copy** = Copy text; describe the text being copied
- **cut** = Cut text; describe the text being cut
- **paste** = Paste text
- **move** = Moving the cursor around in the text editor; specify up, down, left, right and how many times.
- **whitesp** = Creating newlines/white space in the program for readability
- **delete** = Delete text by backspacing; describe the text that is being deleted
- **typing** = Typing text in the editor; (The text is captured in column E)
- **exit** = Exit the text editor;
- **save** = Save the current file in the text editor;

Web/URL Actions

web = Working in a web browser; specify the action

- **brwwith** = Browsing within a content site; clicking a link on a content site that leads to another page in content site
- **brwbetw** = Browsing between content sites = clicking a link on a content site that leads to a new content site.
- **goseng** = Going to a search engine from the problem start page
- **sengsearch** = Doing a search engine search
- **sengresult** = Clicking on a search engine result.
- **sengresultnext** = Browsing to the next (or previous) page of search engine results.
- **sitesearch** = Doing a site search (ie, a within-site search engine search)
- **typurl** = Typing a URL into the browser address bar.
- **back** = Clicking BACK button
- **mouseover** = Mousing over popup and cascading menus or over link text or headers.
- **copy =** Copying info from a web site
- **scan** = Scanning a page
- **scroll** = Scrolling within a page

Col E = keys typed by user

Col F Verbal transcript (from verbal protocol and E notes)

Transcribe **ALL** verbal comments by participant and experimenter;
may do a small bit of paraphrasing as long as you keep all the original meaning;
when in doubt, transcribe every word.
eg, when P is reading aloud from the problem page, do **NOT** code **~~(reading prob page)~~**,
do code the words P is saying Reason: it's important to know what info the P thinks is important enough to read

Segmenting verbal statements correctly is **very important**. Segmentation refers to the decision about how to break the participant's actions and statements into segments that we can code. We can make our segments big (with a lot of actions and statements in each segment) or small (with just one action and just one statement per segment). All of the actions and statements in 1 segment go on 1 line of the excel coding file. We need to make our

segments small enough so that we know exactly what actions & statements our codes apply to. See examples below:

Bad segmenting: because you don’t know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel? Is a sentinal vaue just like null or what? [E "You can use null as an ending, umhum, it is a terminating value"] ok	consider prob (what is the sentinel value) using external info: E consider prob (confusion re sentinel value)

Good segmenting: because you DO know what statements codes refer to

	Action	Transcript	Metacognitive/Programming Strategies
Excel line 1:		Sentinel, what is a sentinel?	consider prob (what is the sentinel value)
Excel line 2:		Is a sentinal vaue just like null or what?	consider prop (confusion re sentinel value)
Excel line 3:		[E "You can use null as an ending, umhum, it is a terminating value"] ok	using external info: E

Integrate info from other sources (eg experimenter’s notes, experimenter’s words) when needed to understand participant’s verbal comments. If P says “This sentence is confusing” and the notes say “P pointed to 2nd sentence of problem page”, include this comment in the transcript, in brackets.

Formatting

Put participant’s words in without brackets.

Use period to indicate end of a thought, even if the thought is incomplete, eg, **Just looking at.**

If P is reading from the problem text or a website put in quotes, eg, **“**Let n represent the actual number of data values**.”**

Put explanatory comments and comments by E (these are not statements made by P) in brackets,

Eg, This sentence is confusing [P pointed to 3rd sentence of page]

Accepted abbreviations (see top of page 1); You can make up your own abbreviations, as long as they are clear.

Col G & H Goals (see problem statement w/ goals and subtasks)

Background: The goals will mostly refer to user statements about the problem, and the subtasks will mostly refer to domain-specific tasks to complete goals from the problem (see the problem statement). For this example, the goals will refer to concepts from the problem; whereas, the subtasks will refer to arrays, loops, products, summing, square root, etc.

Code the goals in column G and the subtasks of goals in column H. In addition, code the start, code, and finish of the goals/subtasks. Code whenever:

- P explicitly **mentions** the **reason for or goal of** a current or planned action **Eg**, "**I am going to start with main",** code **start: begin program** in column G, code **start: construct main** in column H
- You can **infer** the **reason for or goal of** a current or planned action from P's **words, comp actions, and/or keystrokes**; eg, **[P is typing int main(void) {],** code **start: begin program** in column G, code **start: construct main** in column H
- P finishes writing coding for a particular goal or subtask. This is coded in red; **eg**, code **code: begin program** in column G, code **code: construct main** in column H
- P finishes answering a particular goal or subtask after testing and validating to evaluate as ok. This is coded in red; **eg**, code **finish: begin program** in column G, code **finish: construct main** in column H

Eg of: Goals explicitly mentioned by user

Eg 1: user comment="Now I am going to compute and print the square root of the sum of z."

Code=start: compute and print square root in column H

Eg of: Goal/Subtask not explicitly mentioned but can be inferred from user's words and keystrokes

Eg 2: user comment = "If we do the square root then we probably need math." Then the user types

"#include <math.h>"

From the user's words and keystrokes, you can **infer** a goal and subtask.

Code=start: include files in column H

Note: Include any specific details about the goal or answer, as well as any observations made by E, in parenthesis beside the goal. **Eg 3**, start: construct loop (while loop) in column H **OR** **Eg 4,** finish: create arrays (no sentinel value) in column G

Col I Errors in Goals, Subtasks, or Logic

Background: These errors can include syntax, semantic, and suboptimal errors. Code the error, when the error is detected, and when the error is fixed in red in column I.

Code whenever:

- P makes a syntax error in the subtask causing compiler errors; Eg, "that should be it, I think." [P changed stdio.h to stdlib.h], code as deletes needed library (stdio.h) adds unneeded library (include stdlib.h)
- P makes a logical error causing incorrect output; Eg, "Uh, actually 4." [P changed i<5 to i<4 for 5 elements], code as deletes correct loop index ending val (i < 5) add incorrect index ending val (i<4)
- P makes a suboptimal error that doesn't adversely affect the solution; Eg, Prints header inside loop
- P makes a goal/subtask error; Eg, incorrect square root of individual items in z

Cols J & K Metacognitive & Programming Strategies/Schemata can code > 1 on one line

Background: Metacognition (MC) – In Col J, code whenever P makes a statement about his or her own thinking or his or her information search processes while solving a computational problem.

OR

Background: Schema (S) – In Col K, code whenever P makes a statement about his/her answer or makes a statement used in the development of an answer, which may be a program or a verbal solution to a computational problem.

1. **Understand the Problem (MC)** code whenever P thinks about the problem statement.

read prob (MC) code when P explicitly states needing to read or re-read the problem **OR** P reads or re-reads the problem statement or part of it out loud; also note what part of problem read, **eg**, **read prob (1st sentence)**

consider prob (MC) code when P asks a question or makes a statement about the problem statement (these could include statements that a part of the **problem is difficult** or **confusing** or questions about the **next task in the problem**)

eg, "Sentinel, what is a sentinel?", could code: **consider prob (confusion re sentinel value)**

eg, "and now I need to print them out, right. Let's see, where is it?", could code: **consider prob (determine if next goal is to print out arrays)**

2. **Planning (S & MC)** code when makes statements prior or during answering a goal or subtask that usually reflect on P's own knowledge for doing some task.

time planning (MC) code when P discusses general scheduling of activities or how he/she will allocate his/her time and effort

eg, "First, I am going to work on initializing the arrays, and then I will print them out.", code as **time planning**

global planning (MC) code whenever P makes a high-level decision affecting multiple parts of the solution. This includes when P deals with:

- the order of modules/functions in solution **OR**
- the order of working on solution.

eg, "First I have to sum the elements in z, then I can take the square root.", code as **global planning**

eg, "I'm gonna hard code 5 elements in the arrays and get that working, then I'll go back and make it more portable.", code as **global planning**

eg, "I will name them exactly what they are in the program, uh in the problem.", code as **global planning**

eg, "We'll put it in here. If we don't need it, we'll take it out later.", code as **global planning**

eg, "I don't like for loops" [P decides to use while loops throughout solution], code as **global planning**

requirements (S) code when P makes a statement about or recognizes needs and conditions to meet for specific tasks in the solution, taking account of the possibly conflicting requirements. These requirements include any (user and system requirements for the language, domain content, or answer) used to determine solution; also note what is required. **Eg,** "If we do the square root then we probably need math.", code as **requirements (math.h)**

revisit goal/subtask (MC) code whenever P **leaves** a goal or sub-goal/task in working memory and then **returns** to the goal/sub-goal/task – also code what he/she is revisiting in parenthesis, **eg**, **revisit goal/subtask (looping)**

3. Design (S) code when P draws, designs, and/or takes notes on the blank piece of paper; also note what was drawn.
eg, P writes pseudo-code for creating arrays in the solution to a problem before writing the computer code, code as **design (create arrays pseudo-code)**

read design (MC) code when P reads information from the piece of paper containing his/her design, drawings, and/or notes; also note what was read from the design.

consider design (MC) code when P asks a question or makes a statement about his/her design
eg, "", could code: **consider design**

4. **Write code** (S) code when P writes code or mentions wanting to write code for the solution,
eg, **write code:** - also code the details

using prior knowl (S) code whenever P searches memory for relevant prior knowledge or makes statements that use his/her own prior knowledge either before performing a task or during task performance, Note: this includes general statements about the code, considering options for answer, self-questioning/explaining statements about one's answer, goals, sub-goals, code, etc., and low-level planning statements about operations that are possible, postponed or intended – also code the low-level schemata.
eg 1, "I just want to make it initialized by just go up by one, but, um", could code as **using prior knowl (initialize array to contiguous numbers using loop)**
eg 2, "z sub i equals x sub i plus y sub i", could code as **using prior knowl (store product of 2 arrays in 3rd array)**
eg 3, "We can either use a for loop or a while loop to initialize x and y", could code as **using prior knowl (plan loop type)**

need to learn (MC) code when P explicitly states needing to learn or find out more about a particular topic **OR** when P asks a question & uses an external source of information to answer, i.e. help-seeking behavior – also code the topic in parentheses if unclear.
Note: If P asks a question regarding the problem statement, code this under **consider prob OR**
If P asks his/herself a question without using an external source to answer the question, do not code
eg, "what is the syntax for filling up an array?" [P uses the web to find out the syntax], code as **need to learn (array initialization syntax)**.

eg, "I really need to learn how to initialize arrays when I declare them", code as **need to learn (array initialization at time of declaration)**.

recycle goal (MC) code whenever P reuses a goal or sub-goal in working memory. This includes cutting, pasting, and editing code from one goal to answer another goal – also code what he/she is recycling in parenthesis.

confidence (MC) code whenever P makes a clear high or low confidence statement about his/her code or answer. These can include judgment of knowing and feeling of knowing statements qualified by statements such as "If I remember…" – also code details in parenthesis. **Note:** This does not include statements qualified by "I think".

eg, "I know I should use a for loop, but I'm not good at for loops", **confidence (low re for loop)**

eg, "I use loops all the time, and I really know how to use for loops", **confidence (high re knowl about for loops)**

task difficulty (MC) code whenever P mentions the difficulty of some part of their task. However, if P mentions the difficulty of part of the problem/question, code this under **consider prob**,

eg, "this is the same thing I had trouble with a second ago", code as **task difficulty**

learn syntax by guess/compile (MC) code whenever P makes statements about guessing the computer syntax **OR** mentions intentions of using the compiler to determine correctness. This includes statements such as "We'll try it" and "we'll let the compiler catch it".

eg, "probably easier to just test it out and see if, and see what works", code as **learn syntax by guess/compile**

eg, "We'll find out later if it works", code as **learn syntax by guess/compile**

5. **Check/Detect** (S & MC) code whenever P considers his/her answer or performs the act of testing, validating, or detecting errors (or lack of) in the solution. **eg**, **detect:** - also code the details

incremental testing/fixing (MC) code whenever P quickly tests or mentions the need to test small pieces of code as they are added (by any of the methods below) **Note:** This may include commenting out code to see what works OR incrementally fixing errors to detect errors/re-check code

inspection: [own | ext source] (MC) code whenever P checks code via visual inspection and compares it to own knowledge OR knowledge from an external source such as another program, the internet, or a book, i.e. desk-checking. **Note:** Do not use this code when detecting editor errors. This includes when P

- reads computer code he/she has written to check the code or detect errors; also note what was read from the program **OR**
- is evaluating or considering whether some information (eg, a partial answer) is a **complete or correct answer OR**
- quickly notices a logical error (other than typing or editor errors) and quickly changes his/her answer.

eg, "Print the z's, check. There, one more." [P is reading coding and matching parenthesis]; could code **inspection: own (reading print statement & matching parens)**
eg, P is scanning over his/her loop and evaluating whether the loop is correct or complete; could code **inspection: own (evaluating re loop)**
eg, "Actually, it's uh" [P changes loop from i<5 to i<4]; could code **inspection: own (loop stopping value)**

mental execution (MC) code whenever P uses his/her own knowledge to execute existing code mentally **eg**, "that equals one, then that will go to the second thing in the list equals one, so everything in the list will equal one and then,", code as **execute using knowl (predicting effects of code)**

compile (S) code whenever P tests code via compiling it and inspecting the compiler error messages

read compiler (MC) code when P explicitly reads information from the compiler; also note what was read from the compiler.

consider compiler (MC) code when P asks a question or makes a statement about the compiler message **eg**, "uh, for loop [referring to the error message given by the compiler]", could code: **consider compiler (evaluate for loop as error)**

execute using: [output | test-write] (S) code whenever P tests or mentions testing by executing a compiled program and compares the actual and expected output OR uses test-write-statements or a debugging application/environment to inspect internal program data
eg, P executes code after successfully compiling; could code **execute using: output**
eg, P inserts a print statement to check the sum of array elements before taking the square root of the sum, as specified in the problem; could code **execute using: test-write (print sum variable)**

read output (MC) code when P explicitly reads information from the output after executing a program; also note what was read.

consider output (MC) code when P asks a question or makes a statement about the program output by comparing the output to expected output/goals
eg, "good, ok, just what I wanted [output from program x is printed with the values 1 - 10]", could code: **consider output (output matched expected)**

6. Diagnose (S) code when P uses reasoning strategies to identify the cause of incorrect output/error or internal data (eg, backtracking from output code that gave bad output to prior code that provided data to the output code) **OR** uses statements identifying the cause of an error without explicitly stating the reasoning,
eg. "I left out the include statement" [P automatically fixes error w/o inspecting], code as **diagnose:**

inspection: [compiler mssg | prog output] (S) code whenever P checks code via visual inspection and compares it to a compiler message or program output giving an error.

eg, P is scanning over his/her loop and determining the possible error by comparing the code and a compiler message; could code **inspection: compiler mssg (evaluating re loop)**

eg, P is scanning over his/her loop and determining the possible error by comparing the code and the output that was incorrect; could code **inspection: program output (evaluating re print loop)**

debug strategy: [backtracking | elimination | half split] (S) code when P uses a <u>specific approach for finding an error</u> – also code any details about the strategy in parenthesis

eg, "The x and y variables have the correct value, so it must be the z variable causing problems", could code as **debug strategy: elimination (narrow problem to z array)**

hypothesize (MC) code when P poses possible questions and answers about the error he/she is diagnosing – also code the details of the generated hypothesis(es) in parenthesis

eg, "It might be the type of brackets I used in the array declaration causing the error", could code as **diagnose: hypothesize (brackets in array creation)**

7. **Fix code: [correct | incorrect]** (S) code whenever P <u>changes code after detecting an error</u> using some of the above methods – also code whether the repair was correct/incorrect and what was repaired in parenthesis

   eg, [P deletes stdlib.h in program and adds stdio.h], could code as **fix code: correct (change stdlib.h to stdio.h library)**

8. **Use external source: [book | web | old prog | E]** code in Col I whenever P <u>uses an external source of information</u> to help with understanding the problem or answering the problem. In addition, code where the information is coming from, i.e. <u>book, internet, old program, or E</u>. Eg P uses the experimenter to help with a problem, code as **using external source: E.** Note: if P is doing any of the following while using an external source of information, code in Col H:

paraphrasing (MC) code whenever P **summarizes** the gist of some information coming from E, the web, a book, or an old program. Note: if P summarizes part of the problem or question, code this under **consider prob**

support (MC) code when P mentions that 2+ pieces of knowledge **support** or **agree with** or **confirm** or **provide evidence for** each other– also code the 2 pieces of knowledge. Note: this is a positive content evaluation.

eg, P retrieves a potential piece of code from own knowledge but says she needs to verify it; P searches and finds code in a book; then P enters this answer (or accepts previously entered answer as ok). The verifying info in the book would be coded as supporting the potential answer from own knowledge, **using external info: book (support: book verify own)**

lack of support (MC) code when P mentions a **contradiction** or **disagreement** or **lack of support** between 2 or more pieces of knowledge – also code the 2 pieces of knowledge. Note: this is a negative evaluation

site credibility (MC) code whenever P refers to the **credibility** or **believability** or **quality** or **accuracy** of the current external source of info, or mentions **high or low confidence or trust** in the info.

Appendix O Participant 6 Level 2 Problem Behavior Graph

	mcog: goals/recognize schema			---- programming strategy schema/(optional) mcog: strategy ---									---- errors ---	---use extrnl src ---	--- misconceptions ---
s6 Time	Goals	mcog: do next goal in/out seq; switch; revisit	mcog: understand problem/plan	Design (details)	Write Code high low (details)	mcog: low prior knowl	Check-Detect: inspect (mcog)	Check-Detect: compile (details)	Check-Detect: execute (details)	Check-Detect: compare (mcog)	Diagnose (details)	Fix Code (details)	answer actions error; detected; fixes; done fixing; incorrect detect/fix (details)	Use External Source: book; exp; old prog; web	Info re wrong ans Red = completely wrong Green = partially wrong w/ some right
00:32.8	start														
00:48.6		do next goal in seq													
00:52.4	start: Goal 1		square root in prob needs math.h												
01:16.0					high (include math lib)										
01:29.0						include syntax									
01:34.4					high (include stdio lib)										
						other include files									
					high (include stdio lib)										
01:49.0							inspect								
											stdio.h as incorrect and use stdlib.h		incorrect detection		
01:51.2												needed library (stdio.h) adds unneeded library	incorrect fix/error		
01:58.4					high (begin main subroutine; int main() {})										
			paraphrasing, three integer arrays												
02:05.0					write code										
02:18.4			read prob												
		switch													
02:19.2	start: Goal 2					arrays									
02:41.6			what to initialize arrays to												
														exp	
			what to initialize arrays to												
					high (declare and init x & y array at same time)										
03:06.4						arrays									
03:19.6													error: doesn't add sentinel value to x or y array		
03:25.8			z has the product of x and y												
					high (declare array; int z[10])										
03:42.0			read prob												
			the purpose of n												

	mcog: goals/recognize schema			---- programming strategy schema/(optional) mcog: strategy ---									---- errors ---	---use extrnl src ---	--- misconceptions ---
s6 Time	Goals	mcog: do next goal in/out seq; switch; revisit	mcog: understand problem/plan	Design (details)	Write Code high low (details)	mcog: low prior knowl	Check-Detect: inspect (mcog)	Check-Detect: compile (details)	Check-Detect: execute (details)	Check-Detect: compare (mcog)	Diagnose (details)	Fix Code (details)	answer actions error; detected; fixes; done fixing; incorrect detect/fix (details)	Use External Source: book; exp; old prog; web	Info re wrong ans Red = completely wrong Green = partially wrong w/ some right
04:09.0			put it in now & later take out												
					high (declare n variable)										
04:12.0					high (init n variable to number of actual elements)										
04:29.2		do next goal in seq													
04:29.4	start: Goal 3				low (avoids use of for loops)										
04:47.4					high (need counter for while)										
04:51.4					high (while statement)										
05:01.4							inspect								
05:01.6											evaluate 5 as wrong and change to 4		incorrect detection		
												only multiplies 4 of 5 values in the arrays	error		
05:04.8					high (initialize loop counter; array indices start at 0)										
05:21.2						calculating product of arrays									
05:29.0					high (z[i] gets x[i] times y[i] for the product)										
05:47.6					high (increment loop counter at end of loop)										
05:51.0							inspect (code for product looks ok)								
05:51.8		do next goal in seq													
05:52.4	start: Goal 4														
06:13.8					high (use another while)										
06:18.2					high (initialize loop counter; array start at 0)										
06:26.4					low (while statement)								error: only prints 4 of 5 values in the arrays		

	mcog: goals/recognize schema			---- programming strategy schema/(optional) mcog: strategy ---									---- errors ---	---use extrnl src ---	--- misconceptions ---
s6 Time	Goals	mcog: do next goal in/out seq; switch; revisit	mcog: understand problem/plan	Design (details)	Write Code high low (details)	mcog: low prior knowl	Check-Detect: inspect (mcog)	Check-Detect: compile (details)	Check-Detect: execute (details)	Check-Detect: compare (mcog)	Diagnose (details)	Fix Code (details)	answer actions error; detected; fixes; done fixing; incorrect detect/fix (details)	Use External Source: book; exp; old prog; web	Info re wrong ans Red = completely wrong Green = partially wrong w/ some right
06:38.4					high (use text to identify printed numbers)										
08:13.0						print header inside loop							error: prints header inside loop		
08:15.8		switch													
	start: Goal 5 and 6		read last 2 sentences												
08:28.2			order of print and square root												
09:08.4					high (print array values)								error: incorrect fprintf, leaves off f		
09:26.0		revisit													
	revisit: Goal 4					syntax for fprint									
10:19.4		revisit													
	revisit: Goal 5 and 6				high (use square root function and print it)										
10:29.2					high (use square root function and print it)								error: incorrect square root of individual items in z		
		revisit													
	revisit: Goal 4						inspect								
10:45.2					high (close parens and add newline)										
10:55.6							inspect								
											header in wrong place, inside loop		detected: header inside loop error		
11:49.4												take header out of loop	error: breaks up a print statement w/ tab		
11:56.6												put header at top of loop	fixes: header inside loop error		
12:04.4					high (increment loop counter at end of loop)								error: leaves off semicolon		
12:16.4							inspect								
12:21.6					high ()										
12:26.6							inspect (checking that brackets {} match)								
12:47.6		revisit													
12:49.4	revisit: Goal 1				high (return from main)										
12:57.2						quality of the program									
13:06.6								compile							
13:15.6											left out lib		detected: library error (left out stdio.h)		
13:25.8												adds stdio.h	fixes: adds library		
13:44.6								compile							

	mcog: goals/recognize schema			.---- programming strategy schema/(optional) mcog: strategy ---									.---- errors ---	.---use extrnl src ---	.--- misconceptions ---
s6 Time	Goals	mcog: do next goal in/out seq; switch; revisit	mcog: understand problem/plan	Design (details)	Write Code high low (details)	mcog: low prior knowl	Check-Detect: inspect (mcog)	Check-Detect: compile (details)	Check-Detect: execute (details)	Check-Detect: compare (mcog)	Diagnose (details)	Fix Code (details)	answer actions error; detected; fixes; done fixing; incorrect detect/fix (details)	Use External Source: book; exp; old prog; web	Info re wrong ans Red = completely wrong Green = partially wrong w/ some right
14:14.4											on line 26 and				
14:24.6											need to backspace		detected: print statement on two lines error		
14:33.0												print statement on two lines	fixes: print statement on two lines error		
14:40.2							inspect								
15:06.4								compile (less errors and error about print stmt)							
15:07.4											left out f on print statement		detected: fprintf error		
15:21.2												fprint error	fixes: fprinf error		
15:24.0								compile (error on line 33 & didn't use n)							
15:39.8											didn't use n		detected: didn't use n & line 33		
16:01.2												delete n	fixes: deletes n variable		
16:05.8								compile (expected semicolon)							
16:14.2											missing semicolon		detected: missing semicolon		
16:28.4												add semicolon	fixes: added semicolon		
16:31.6								compile							
16:35.2									execute (formatting and square root of elements)						
17:01.2									execute (check w/ calulator)						
17:52.4											(only formatting errors)				
17:55.0													error: all the code is lined up on the left but he is worried about output formatting	exp	
17:59.4										compare					
18:11.0			(read prob); (all static not dynamic in problem)												
										compare					
							inspect								
			read prob												
			do the fastest												
			read prob												
										compare					
											read prob wrong				
19:40.2										compare					

	mcog: goals/recognize schema			.---- programming strategy schema/(optional) mcog: strategy ---									.---- errors ---	.---use extrnl src ---	.--- misconceptions ---
s6 Time	Goals	mcog: do next goal in/out seq; switch; revisit	mcog: understand problem/plan	Design (details)	Write Code high low (details)	mcog: low prior knowl	Check-Detect: inspect (mcog)	Check-Detect: compile (details)	Check-Detect: execute (details)	Check-Detect: compare (mcog)	Diagnose (details)	Fix Code (details)	answer actions error; detected; fixes; done fixing; incorrect detect/fix (details)	Use External Source: book; exp; old prog; web	Info re wrong ans Red = completely wrong Green = partially wrong w/ some right
19:42.6											not square root of each item, but the sum of items		detected: square root of individual items in z		
19:45.4		revisit													
19:45.6	revisit: Goal 5											delete square root of each element			
20:08.8												delete square root of each element	fixes: delete square root of each element		
20:12.2					high (need separate loop or not)								error: does not delete the %lf in print statement		
20:22.2					high (use another loop)										
20:23.8					low (construct while loop)										
		revisit													
20:28.4	revisit: Goal 2				high (need a sum variable)										
20:35.4					high (declare sum var)										
20:39.2		revisit													
20:42.6	revisit: Goal 6				low (the need for 4 as the loop ending value)								error: there are 5 elements not 4		
20:48.8					high (need to reinitialize I for while loop)										
20:56.0					high (open while stmt)										
21:03.8					high (calculate the sum of values in z array)										
21:19.4							inspect								
21:22.6					low (close while stmt w/o incrementing loop counter)								error: does not increment the loop counter		
		revisit													
21:25.0	revisit: Goal 2				high (need to initialize sum to zero)										
21:28.6								inspect							
21:40.0		revisit													
21:41.6	revisit: Goal 6				high (construct a print statement)										
21:55.8		revisit													
21:56.6	revisit: Goal 5				high (use square root function)								error: square root function is capital		
21:59.0		revisit													
22:01.4	revisit: Goal 6						inspect								

	mcog: goals/recognize schema			.---- programming strategy schema/(optional) mcog: strategy ---									.---- errors ---	.---use extrnl src ---	.--- misconceptions ---
s6 Time	Goals	mcog: do next goal in/out seq; switch; revisit	mcog: understand problem/plan	Design (details)	Write Code high low (details)	mcog: low prior knowl	Check-Detect: inspect (mcog)	Check-Detect: compile (details)	Check-Detect: execute (details)	Check-Detect: compare (mcog)	Diagnose (details)	Fix Code (details)	answer actions error; detected; fixes; done fixing; incorrect detect/fix (details)	Use External Source: book; exp; old prog; web	Info re wrong ans Red = completely wrong Green = partially wrong w/ some right
					high (finish print statement)										
22:12.8								compile							
22:44.4											looking for lines from compiler				
22:52.8											took out arg so need to take out the formatting for it		detected: %lf in print statement for arrays		
22:58.0												print stmt for arrays	fixes: print statement for arrays		
23:01.8											capitial S used in square root		detected: square root function		
23:04.4												square root function	fixes: square root function		
23:07.6								compile							
23:09.8									execute (no print of square root)						
23:27.8											endless loop				
23:36.6											didn't increment i		detected: did not increment loop counter		
23:38.6												increment i			
23:40.8							inspect								
23:45.0												finish incrementing i	fixes: increment loop counter		
23:51.2								compile							
23:53.4									execute						
24:05.0									execute (check output to calculator)						
24:45.4										compare					
24:51.0			read prob												
										compare					
			what about uneven arrays												
														exp	
			arrays always even												
26:46.6										compare					

Appendix P Participant 6 Level 3 Model Results

S6	Extnl Src	Metacognition				Strategies/Schemata				
		Control Processes		Monitor Processes		Design	Write Code	Check-Detect	Diagnose	Fix Code
00:48.6		do next goal in seq								
00:52.4				square root in prob needs math.h						
01:16.0							high (include math lib)			
01:29.0			include syntax							
01:34.4			other include files							
							high (include stdio lib)			
01:49.0					inspect					
									evaluate stdio.h as incorrect and use stdlib.h	
01:51.2										needed library (stdio.h) adds unneeded library
01:54.6										
01:58.4							high (begin main subroutine; int main() ()			
				paraphrasing, three integer arrays						
02:05.0							write code			
02:18.4				read prob						
		switch								
02:19.2			arrays							
02:41.6				what to initialize arrays to						
	exp									
				what to initialize arrays to						
							high (declare and init x & y array at same time)			
03:06.4			arrays							
03:19.6										
03:25.8				z has the product of x and y						
							array; int z[10])			
03:42.0				read prob						
				the purpose of n						
04:09.0				put it in now & later take out						
							high (declare n variable)			
04:12.0							high (init n variable to number of actual elements)			
04:29.2		do next goal in seq								
04:29.4							low (avoids use of for loops)			
04:47.4							high (need counter for while)			
04:51.4							high (while statement)			
05:01.4					inspect					
05:01.6									evaluate 5 as wrong and change to 4	
										only multiplies 4 of 5 values in the arrays
05:04.8							high (initialize loop counter; array indices start at 0)			
05:21.2			product of arrays							
05:29.0							high (z[i] gets x[i] times y[i] for the product)			
05:47.6							(increment loop counter at end of loop)			
05:51.0					inspect (code for product looks ok)					
05:51.8		do next goal in seq								
05:52.4										
06:13.8							another while)			
06:18.2							high (initialize loop counter; array start at 0)			
06:26.4							low (while statement)			
06:38.4							high (use text to identify printed numbers)			
08:13.0			header inside loop							
08:15.8		switch								
				read last 2 sentences						
08:28.2				order of print and square root						
09:08.4							high (print array values)			
09:26.0		revisit								
			syntax for fprint							
10:19.4		revisit								
							high (use square root function and print it)			
10:29.2							high (use square root function and print it)			
		revisit								
					inspect					

S6	Extnl Src	Metacognition				Strategies/Schemata				
		Control Processes		Monitor Processes		Design	Write Code	Check-Detect	Diagnose	Fix Code
10:45.2							high (close parens and add newline)			
10:55.6					inspect					
									header in wrong place, inside loop	
11:49.4										take header out of loop
11:56.6										put header at top of loop
12:04.4							(increment loop counter at end of loop)			
12:16.4					inspect					
12:21.6							high ()			
12:26.6					inspect (checking that brackets {} match)					
12:47.6		revisit								
12:49.4							high (return from main)			
12:57.2			the program							
13:06.6								compile		
13:15.6									left out lib	
13:25.8										adds stdio.h
13:44.6								compile		
14:14.4									on line 26 and	
14:24.6									need to backspace	
14:33.0										print statement on two lines
14:40.2					inspect					
15:06.4								compile (less errors and error about print stmt)		
15:07.4									left out f on print statement	
15:21.2										fprint error
15:24.0								compile (error on line 33 & didn't use n)		
15:39.8									didn't use n	
16:01.2										delete n
16:05.8								compile (expected semicolon)		
16:14.2									missing semicolon	
16:28.4										add semicolon
16:31.6								compile		
16:35.2								execute (formatting and square root of elements)		
17:01.2								execute (check w/ calulator)		
17:52.4									(only formatting errors)	
17:55.0	exp									
17:59.4					compare					
18:11.0				(read prob), (all static not dynamic in problem)						
					compare					
					inspect					
				read prob						
				do the fastest						
				read prob						
					compare					
									read prob wrong	
19:40.2					compare					
19:42.6									not square root of each item, but the sum of items	
19:45.4		revisit								
19:45.6										delete square root of each element
20:08.8										delete square root of each element
20:12.2							high (need separate loop or not)			
20:22.2							high (use another loop)			
20:23.8							(construct while loop)			
		revisit								
20:28.4							high (need a sum variable)			
20:35.4							high (declare sum var)			
20:39.2		revisit								
20:42.6							low (the need for 4 as the loop ending value)			
20:48.8							high (need to reinitialize I for while loop)			
20:56.0							high (open while stmt)			
21:03.8							high (calculate the sum of values in z array)			
21:19.4					inspect					
21:22.6							low (close while stmt w/o incrementing loop counter)			
		revisit								

S6	Extnl Src	Metacognition				Strategies/Schemata				
		Control Processes		Monitor Processes		Design	Write Code	Check-Detect	Diagnose	Fix Code
21:25.0							high (need to initialize sum to zero)			
21:28.6					inspect					
21:40.0		revisit								
21:41.6							high (construct a print statement)			
21:55.8		revisit								
21:56.6							high (use square root function)			
21:59.0		revisit								
22:01.4					inspect					
							high (finish print statement)			
22:12.8								compile		
22:44.4									looking for lines from compiler	
22:52.8									take out the formatting for it	
22:58.0										print stmt for arrays
23:01.8									capitial S used in square root	
23:04.4										square root function
23:07.6								compile		
23:09.8								print of square root)		
23:27.8									endless loop	
23:36.6									didn't increment i	
23:38.6										increment i
23:40.8					inspect					
23:45.0										finish incrementing i
23:51.2								compile		
23:53.4								execute		
24:05.0								execute (check output to calculator)		
24:45.4					compare					
24:51.0				read prob						
					compare					
				what about uneven arrays						
	exp									
				always even						
					compare					
26:46.6		finish								

Appendix Q Raw Observations & Off-line Interviews with 11 Participants

Subject #1 102 Student He used design on paper first, then found a program that he had written that somewhat matched what he wanted to do, but he got himself stuck by going that direction. Next, he went in a different direction, and wondered if he needed two loops. He then abandoned that idea and tried compiling the program, but couldn't figure out the errors. He knew the error was on line 9, where the for loop was; so he used the web to review for loops. He saw that the logic behind his for loop was wrong, so he tried fixing that, and then still failed. Now he goes back to original idea, and get the thing to compile. However, now he is stumped by the output! At this point it has been 45 minutes...

Subject #2 102 Student No design on paper at any point. He tried to find an array program in his directory, but then went to google to remember how to do an array. Began writing the program, but didn't follow the syntax given in the google example! Continues to write code, and looks up for loops in google to construct that. He never initializes the sum variable that he uses w/o a value (he better be glad that it was automatically initialized to zero!). Then he goes to the second part of the question, the square root stuff, w/o compiling, running, testing the first bit of code!!! (this is a good reason to keep this in here). After it doesn't compile, he comments out the square root stuff and works on the first part. Uses google to analyze the arrays again, then changes the declaration, but doesn't understand what he is really doing because what he codes doesn't match what he is saying. Once it gives output, he assumes it works correctly! Then he focuses on the square root, but doesn't work because gcc can't find math.h. I ask if he wants to use a different compiler, and he says that he can't because gcc gives him warnings and error messages! This is when we stopped at 45 minutes...!

Subject #3 102 Student He began by re-reading the problem statement (which they all did). He wrote the variables he was interested in capturing on paper before coding. Then he went to coding. Began with writing comments (the first to do that!), and declares x and y arrays, but the first to think about room for a sentinel value. Thinks about portability and asking a user vs. hard-coding the elements of the array; however, he hard-codes the n vs. using the sentinel value to find the n, which would make it way more portable. Then he considers a while versus a for loop and chooses a while loop because some flaw in thinking about conditionals and for loops. Then he wonders if he needs two/nested loops, but no. He also does the square root stuff before ever compiling and running to make sure the first part is right. Debugging: He then uses other programs to check on main syntax, doesn't look up anything on the web and uses own work around knowledge to get the arrays initialized and working, then realizes he doesn't increment the variable in his while loop (that he really should have been using a for loop!). His program worked and he did a good job, minus some misconceptions. This is when we stopped at 45 minutes...!

In this example, he never tested the output with different input, but he did analyze the output! He recognized the sentinel value and defined it as -1. Instead of finding the n value, which would be more portable, he hard coded this value and made it harder to change, i.e. what is the purpose of the sentinel value at this point.

Subject #4 212 Student He began by re-reading the problem. He opened up the terminal and began coding. He declared three, x, y, and z, arrays to hold a max of 10 elements, and then he initialized the x and y arrays to two elements (another who initialized the elements outside the declaration, i.e. x[0]= and x[1]=.) But then it got interesting...

He decided not to use any loops because he said, "it would take longer to construct for loops, and since we are hard-coding the array values, then we can just write out the array multiplication and printing." Therefore, he decided to populate two elements in x and y, then he hard-coded the two elements of z to equal x[]*y[]. Then of course he hard coded the print statements for both elements rather than using a for loop... He compiles, fixes things, analyzes his two elements and the first part of the problem, then goes to the second part. He hard-codes the sum of the two elements in z and prints the square root. He compiles, executes, analyzes, and says he is done...

So, I ask him, "what happens if he wants to test the array with 8 elements." I proceeded to tell him, "I know that you said this was easy and you were only using two elements out of the 10, but what happens if you do want to add add more elements and how much code would you have to change." He told me, "Well, I am already in there to hard-code elements in the array, so I can just add more code, cut and paste".

At this point I got really discouraged and stopped recording b/c the "verbal protocol" isn't about changing the student's mind! After I stopped recording and went into teacher mode, I asked, "what happens if he wants to change his code to get input from the user or from a file?", and he answered, "well, if this were for a class and a I had a few days, then I would make it more robust. I would have created for loops and put the arithmetic and prints in loops." I asked what he thought the purpose of the sentinel value and n value where, and he replied, ”Oh yeah, I meant to look up sentinel value. I didn’t know what that was, and then I got forgot about that and when I re-read the problem I didn’t even see that.”

He never looked up anything on the web, book, or did any kind of design on paper.

After we stopped recording, he expressed his concerns about the raytracing application in all his classes because he doesn’t feel he knows how to apply anything outside of raytracing. His biggest concern seems to be with the data structures. I guess he feels like he isn’t getting a complete view of cs, and he said several times that he thought graphics would be cool to do and study at age 10, but now that he is older he wants to know more about his options in cs.

Subject #5 102 Student The subject seemed to be having trouble understanding the problem and what z should equal. It became evident when he started writing the program, but he continued to re-read the problem to understand and analyze his solution.

He began by re-reading the problem. He opened up the terminal and began coding. He declared an x and y array arrays with no maximum elements, and then he initialized the x and y arrays at the time of declaration with three elements. Then he declares the n variable.

He uses a while loop to find n by counting the elements in x that are not equal to NULL. Then he does this for y and sums the number of elements in x and y. He re-reads the problem, and by the example given in the problem, he realizes he is wrong and corrects his program.

Then he uses a for loop to find elements in z, but he declares another variable to increment through the arrays. He realizes he needs to declare z as he writes the statement to fill elements of z in the loop. He uses ¡= in loop and populates one too many elements in z, which he never fixes.

He creates another for loop to print the three arrays, and declares another variable to increment through arrays!!!

He compiled, figured out errors, re-compiled, until it ran correctly. Then, he worked on the second part of the problem. Wondered if he could print the square root and do everything in the loop above when he printed x, y, z, but he decided to create another for loop for the summation of elements in z. Again, he declares yet another increment variable (he has run out of i, j, k, and now he uses d)!!!!

He never designed on paper, and he was apprehensive of looking things up on the web at first. Then, he ignored what it said because he assigned the value from sqrt to an integer.

He compiles, gets all the output, but never tests with different numbers or analyzes the output.

He didn't verbalize much (but he did talk more than the first subject), and his ethnicity was Latin-American. After recording, he said that he was a little nervous and that might have made him not do some things.

Interesting observation, but no one really uses ++ for incrementing by one.

Subject #6 102 Student He began by re-reading the problem, and then opened up his terminal and began coding. He never wrote anything down on paper, and he never used an external source of information. Instead, he would say things like, "well, I know I should use a for loop, but I don't like for loops (laugh) So, I am just going to use whiles."

He began his code by including math.h because of the square root function (most people ignored this part of the problem until later).

When he goes to create his while loop, he decides to have an increment variable, i, and then he decides to hard code the conditional value, instead of using the n value (which he also hard coded instead of using a sentinel value to find the number of actual data values). He did use i++, which no one else is using.

He creates another while loop to print the arrays, and then starts on the second part of the problem. He wants to print and do the second part in the same loop where he prints so that he doesn't have to create another loop. He does this and takes the square root of each element in z (which is not the problem). He compiles, fixes (however at one point he is looking for the wrong thing, but corrects the right thing. The compiler said he was missing a ; before a , and then he was looking for pairs???)... He re-compiles, executes, and he actually analyzes his output with a calculator!! But, he doesn't test with other data values. He acts like he is done, so I ask him if he is, and he says, "well, let me re-read the problem to make sure I am doing everything right."

That is when he realized he did the second part of the problem wrong. Creates another while loop to find the sum of the elements in z and takes the square root of the sum. However, he has forgotten to put i++ in his while; so, when he compiles and runs it is in an infinite loop. However, he fixes it, gets it running again with the right output. Whew....

In the interview after the recording, I find out that he doesn't like for loops because he knows several different programming languages and gets the syntax all wrong in C, and he already knows while loops. I asked him that if he knew he should use a for loop and have access to books and the internet, why wouldn't you look it up? He shrugged his shoulders:)

I told him that kind of thinking will get him into trouble later:) I told him that it already got him into trouble when he had the infinite loop b/c he forgot the i++ in his last while loop. If he had of used the for loop, chances are that he wouldn't have forgotten that because the syntax of the for loop lends itself to remember: initialize, test, increment...

His metacognition was very high, i.e. he knew when he should and shouldn't be doing things but he has learned bad habits because he is self taught from 7th grade! Also, he has been my youngest subject, 18.

Subject #7 212 Student Oh goodness, goodness, there is a pattern resurrecting itself!!!

Another student who said he knew he should use a for loop, but he doesn't like them!!! So, of course he wrote two while loops, and when he got himself into trouble with the second while he used a for loop for the second part of the problem. (which he constructed and used just fine!) So, now one question I have is ”why are they so scared of the for loop?” This is a metacognitive question about confidence. Maybe they need more practice to become confident (and I would say the same about arrays!!!).

Also, he too had issues with initializing his array. He wanted to do it separate from the declaration, but wanted to initialize all elements in one line by setting the array name to the values, i.e. x = 1, 2, 3; He didn't understand why this didn't work, so he moved the initialization up to declaration (which many have done this orjust separated the elements, i.e. x[0]=1; x[1]=2; x[2]=3;Yes, this is what you are supposed to do it, but I found out through these verbal protocols that they don't know why it doesn't work and why what they did fixes it, i.e. schema isn't fully developed for arrays and their addressing.

He did look up arrays on the web but he got himself into a pickle when he didn't realize he was looking at C#, which I had noticed immediately and then had to tell him.

He again didn't design or write anything down, and he did solve the problem incorrectly because he also took the square root of each element in z and ignored the sum. I asked him once if he was done and he re-read the problem and said, ”oh no, I think I need doubles because it doesn't say anything about the type.” So he fixed that, and then when I asked him again if he was done he said yes. So, I said, ”ok, will you re-read the second part of the problem”.And then he realized what he did wrong, but at this point it would just be watching him change his code a bit.

Subject #8 212 Student He did the best job so far, but I really had to remind him to think out loud. He told me in the end that if the problem had been harder, then he would have talked more.

Anyway, he still ignored the sentinel value. I think there have only been two people who have accounted for the value.

He didn't design or write anything down on paper, but he did solve the problem incrementally and kept going back to the problem statement to check to make sure he was doing things right and move forward to the next step.

He got as far as writing, whil, then backspaced and used nothing but for loops. He was the first to populate the arrays using a loop, and used an n value for his loops, which was hard coded and not found by the sentinel value. He got down to the end of his program and realized that he forgot, int main() .

Of course I noticed it when he first started out, but I wondered why he noticed it way at the end and w/o seeing the top of his program or compiling it. After the recording I asked this and he told me because he is anal about indenting and he noticed all his for loops were all the way left and they should be in a few spaces. I said that was interesting that he says that b/c I have always thought programming requires attention to detail and a little anal retentiveness:)

He never looked anything up, and he seemed to be pretty bright. When asked about the sentinel value, he did understand it but just forgot about it.

Subject #9 212 Student He begins by asking me what a sentinel value is (so he does know what it means, but then never uses it). After the recording I asked why he didn't use it, and he said, ”oh yeah, well, I kind of forgot about it”. I told him that I thought that was interesting because Dr. Gugerty (and the psych world) say that people tend to redesign the problem when problem solving, and they do this for various reasons. It is fine if someone has thought long and hard about the problem and redesigns it to more efficiently and effectively meet the goal. However, it is something quite different when people redesign the problem to fit to their needs, and there are many reasons for this!

Anyway, he starts by trying to find an old program with arrays, but it becomes too much hassle, so he just recalls from memory. He populates his lists using a for loop, and gets very confused about how to set the elements to 1, 2, 3, 4, etc. He finally declares another variable, j, that is used to assign the values (still not sure why he didn't use the i and n that he already has!). Oh yeah, and he assigns all 10 values (not just a few to test with!).

However, I must say that he was the first to incrementally write, compile, and execute his code as he programmed versus writing the whole thing (or most of it) and having lots of errors. I encouraged this behavior at the end:)

The only other strange thing was that he fumbled for a while on how to sum the elements in z, but then he finally got it.

He returned the square root to an integer variable, but he did notice that his answer was rounded. However, he didn't do anything about it, but that isn't explicitly stated by the problem either.

Subject #10 212 Student This was the first female participant. She began by adding the whole main template with return statement, and then declared 3 arrays of size 10. She asks about the total number of values in the arrays, and the experimenter tells her that she can choose the number. She chooses 6 and sets n value, and uses a while loop to populate the x and y arrays for 6 values. She considers NULL as a sentinel value, but then changes to use 0 as sentinel because NULL is a character. However, she puts the sentinel at the last element rather than where the last value is, i.e. x[6] and y[6]. Next, she determines that the arrays must be initialized to 0 before populating them, or it will seg fault. This is not correct thinking! Because of this, she changes the sentinel value to 15 for an unjustified reason because 15 could be a number in the array.

Consistently she uses the wrong variable for the index values in her while loops, but she diagnoses and fixes these errors later. However, she writes the whole program before compiling or debugging. She never uses the sentinel value and only uses while loops. She sums and prints array elements that dont have values in them, and she doesnt change her loops to increment through 6 elements that she populated instead of all 10.

Throughout the verbal protocol and after, she expressed being burned out from writing a ray tracing program. She never seemed to understand completely why she should not have used all 10 elements when only initializing 6 elements and needed another loop to initialize 10 elements to zero before populating the arrays. The experimenter explain that a seg fault happens when accessing an element that is not there, i.e. going outside memory which is why it sometimes changes when more lines are added or subtracted from the program.

Subject #11 212 Student He began by rereading the problem. He ignores the sentinel value in the problem, and he uses a for loop to populate all 10 elements in his arrays. He makes a lot of syntax errors in his for loops such as commas instead of semicolons and switching the position of the conditional statement and incrementing the index. However, he does find all these errors by inspecting his code or compiling. He struggles with summing the elements in a loop, so he hard codes summing the elements in z.

He overlooks the loop error where he increments before checking the condition several times, even after compiling and recompiling, before the experimenter gave a hint about the where to look. In addition, the experimenter reminds him to think about the compiler in order to remember to use lm with the compiler, which he does remember unlike subject #2. He seems to make numerous little errors, and he acts very confused and tired. He is the first one to not put any library files until the end after compiler errors, and he doesnt compile until after writing the whole program. Like most students, he doesnt change values to check his answer. He accepts his answer based on appearance.

After the verbal protocol, he expressed how tired he was from the ray tracing program that #10 was tired from writing. He says that he knew how to sum the items in a loop, but just couldn't think of it because he was tired.

Bibliography

[1] P. Adey, M. Shayer, and C. Yates. *Thinking science: Student and teachers materials for the CASE Intervention.* Nelson Thornes, London, 3rd edition, 2001.

[2] Esma Aimeur, Claude Frasson, and Michel Lalonde. The role of conflicts in the learning process. *SIGCUE Outlook*, 27(2):12–27, 2001.

[3] Shereef Abu Al-Maati and Abdul Aziz Boujarwah. Literate software development. *Journal for Computing Sciences in Colleges*, 18(2):278–289, December 2002.

[4] Brenda Allen and Eleanor Armour-Thomas. Construct validation of metacognition. *The Journal of Psychology*, 127(2):203–211, 1991.

[5] Vicki L. Almstrum, Peter B. Henderson, Valerie Harvey, Cinda Heeren, William Marion, Charles Riedesel, Leen-Kiat Soh, and Allison Elliott Tew. Concept inventories in computer science for the topic discrete mathematics. *SIGCSE Bull.*, 38(4):132–145, 2006.

[6] Vicki L. Almstrum, David Klappholz, Steven Condly, and John Clement. Are they learning what (we think) we're teaching? In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 195–195, Houston, Texas, USA, March 2006. ACM Press.

[7] J. Anderson. *The Architecture of Cognition.* Harvard University Press., Cambridge, MA, 1983.

[8] John R. Anderson, James G. Greeno, and Paul J. Kline. *Acquisition of Problem-Solving Skills*, chapter 6, pages 191–230. Lawrence Erlbaum Associates, Inc, Hillsdale, N.J., 1981.

[9] L. W. Anderson and D. R. Krathwohl. *A taxonomy for learning, teaching and assessing: A revision of Bloom's Taxonomy of educational objectives: Complete edition.* Longman, New York, 2001.

[10] Anonymous. An update on learning styles/cognitive styles research. *The Teaching Professor*, 19(1):3, January 2005.

[11] Eleanor Armour-Thomas and Norris M Haynes. Assessment of metacognition in problem solving. *Journal of Instructional Psychology*, 15(3):87–93, September 1988.

[12] Karina V. Assiter. Accessibility of analysis of algorithms: From programming to problem solving. *Journal for Computing Sciences in Colleges*, 21(2):185–194, December 2005.

[13] David P. Ausubel. *Educational psychology; a cognitive view.* Holt, Rinehart and Winston, New York, 1968.

[14] Roger Azevedo, John T. Guthrie, and Diane Seibert. The role of self-regulated learning in fostering students' conceptual understanding of complex systems with hypermedia. *Journal for Educational Computing Research*, 30(1 & 2):87–111, 2004.

[15] L. Baker and A. L. Brown. *Metacognition skills and reading*, pages 353–394. Longmans, Green, New York, 1984.

[16] Sir Frederic Bartlett. *Thinking: An Experimental and Social Study.* Basic Books, Inc, New York, 1958.

[17] Sir Frederic C. Bartlett. *Remembering: A study in experimental and social psychology.* The University Press, Cambridge, MA, 1932.

[18] Jonathan Baxter. A bayesian/information theoretic model of bias learning. In *COLT '96: Proceedings of the ninth annual conference on Computational learning theory*, pages 77–88, New York, NY, USA, June 1996. ACM Press.

[19] Beryl Lieff Benderly. Every day intuition. *Psychology Today*, 23(9):35–39, September 1989.

[20] B. S. Bloom and Lois J. Broder. *Problem-solving Processes of College Students: Ana Exploratory Investigation.* The University of Chicago Press, 1950.

[21] B. S. Bloom and D. R. Krathwohl. *Taxonomy of educational objectives: The classification of educational goals, by a committee of college and university examiners. Handbook 1: Cognitive domain.* David McKay Company, Inc., New York, 1956.

[22] Toni L. Blum and Wayne J. Staats. The assessment of cognitive skills for computer science education. Technical report, 29th ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico, November 1999.

[23] National Science Board. *Higher Education in Science and Engineering*, chapter 2, pages 1–39. National Science Foundation, http://www.nsf.gov/statistics/seind06/pdf/c02.pdf, 2006.

[24] George M. Bodner. Constructivism: A theory of knowledge. *Journal of Chemical Education*, 63(10):873–878, October 1986.

[25] Grady Booch. *Object-oriented Analysis and Design.* Benjamin/Cummings, Redwood Cita, CA, 2nd edition, 1994.

[26] M. Bower. A taxonomy of task types in computing. In *Proceedings of the 13th Annual Conference on innovation and Technology in Computer Science Education*, June 2008.

[27] John D. Bransford and Barry S. Stein. *The Ideal Problem Solver.* W. H. Freeman and Company, New York, 2nd edition, 1993.

[28] Grant Braught, Craig S. Miller, and David Reed. Core empirical concepts and skills for computer science. *SIGCSE 2004*, pages 245–249, March 2004.

[29] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Computing with Production Systems*, chapter 1, pages 3–31. Addison-Wesley Publishing Co., Inc, 1985.

[30] Jerome S. Bruner. *Readiness for Learning*, chapter 23, pages 413–425. Harvard University Press, Cambridge, MA, 1960.

[31] Jerome S. Bruner. *Beyond the information given; studies in the psychology of knowing.* Norton, New York, 1973.

[32] Jerome S. Bruner. *Acts of meaning.* Harvard University Press, Cambridge, MA, 1990.

[33] Patricia F. Campbell and George P. McCabe. Predicting the success of freshman in a computer science major. *Communications of the ACM*, 27(11):1108–1113, November 1984.

[34] Zhengxin Chen. Building expert systems through the integration of mental models. *ACM*, pages 754–761, 1988.

[35] Perry C. Cheng, Danny Kilis, and Graeme Knight. Knowledge assessment using fuzzy conceptual representation. In *SAC '97: Proceedings of the 1997 ACM symposium on Applied computing*, pages 3–9, New York, NY, USA, 1997. ACM Press.

[36] Michelene T. Chi, Paul J. Feltovich, and Robert Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5:121–152, 1981.

[37] D. Chinn, C. Spencer, and K. Martin. Problem solving and student performance in data structures and algorithms. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, June 2007.

[38] Ashraful A. Chowdhury, C. Van Nelson, Clinton P. Fuelling, and Roy L. McCormick. Predicting success of a beginning computer course using logistic regression (abstract only). In *CSC '87: Proceedings of the 15th annual conference on Computer Science*, page 449, New York, NY, USA, 1987. ACM Press.

[39] Chris. *Reasoning: Mental Models.* Mixing Memory, http://mixingmemory.blogspot.com/2004/12/reasoning-mental-models.html, December 19 1994.

[40] Michael J. Clancy and Marcia C. Linn. Patterns and pedagogy. *SIGCSE 1999*, March 1999.

[41] John Clement. Students' preconceptions in introductory mechanics. *American Journal of Physics*, 50:66–71, 1982.

[42] John Clement. *A Conceptual Model Discussed by Galileo and Used Intuitively by Physics Students.*, chapter 14, pages 325 – 340. L. Erlbaum Associates, 1983.

[43] John M. Clement. A call for action (research): Applying science education research to computer science instruction. *Computer Science Education*, 14(4):343–364, 2004.

[44] Douglas H. Clements. Effects of logo and cai environments on cognition and creativity. *Journal of Educational Psychology*, 78:309–318, 1986.

[45] Douglas H. Clements and Bonnie K. Nastasi. Measurement of metacomponential processing in young children. *Psychology in the Schools*, 24(4):315–322, 1987.

[46] Andrew M. Colman. *A Dictionary of Psychology.* Oxford University Press, http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t87.e7017, 2nd edition, 2006.

[47] Andrew M. Colman. *A Dictionary of Psychology: constructivism.* Oxford University Press, http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t87.e1823, 2006.

[48] Jack Copeland. Turing's o-machines, penrose, searle, and the brain. *Analysis*, pages 128–138, 1998.

[49] John Corcoran. *Schema.* Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/schema/, May 2004.

[50] National Research Council. *How People Learn: Brain, Mind, Experience, and School.* National Academy Press, Washington, 2000.

[51] National Research Council. *Knowing what Student Know: The Science and Design of Educational Assessment*. National Academy Press, Washington, September 2001.

[52] National Research Council. *Evaluating and Improving Undergraduate Education: In Science, Technology, Engineering, and Mathematics*. The National Academies Press, Washington, D.C., 2003.

[53] K. J. W. Craik. *The Nature of Explanation*. Cambridge University Press, 1943.

[54] Donald W. Dearholt and Roger W. Schvaneveldt. *Pathfinder Associative Networks: Studies in Knowledge Organization*. Ablex Publishing Corporation, 1990.

[55] Peter J. Denning. A debate on teaching computing science. *Communications of the ACM*, 32(12):1397–1414, December 1989.

[56] Dr. Robert B.K. Dewar and Dr. Edmond Schonberg. Computer science education: Where are the software engineers of tomorrow? *CrossTalk: The Journal of Defense Software Engineering*, 21(1):28–30, January 2008.

[57] J. Dewey. *How we think*. DC Heath and Company, Boston, MA, 1910.

[58] Linda P. DuHadway, Stephen W. Clyde, Mimi M. Recker, and Donald H. Cooley. A concept-first approach for an introductory computer science course. *Journal for Computing Sciences in Colleges*, 18(2):6–16, 2002.

[59] F. T. Durso, K. Rawson, and S. Girotto. *Comprehension and situation awareness*, chapter 7, pages 163–193. Wiley, 2nd edition, 2007.

[60] Caroline Brayer Ebby. The powers and pitfalls of algorithmic knowledge: a case study. *Journal of Mathematical Behavior*, 24:73–87, 2005.

[61] Aaron Enright, MaryAnn Robbert, Randall Stafford, Kathryn Wilkens, and Linda Wilkens. Staying in sync with industry needs. *Journal for Computing Sciences in Colleges*, 15(5):32–35, May 2000.

[62] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, Cambridge, MA, 1993.

[63] S. Fitzgerald, B. Simon, and L. Thomas. Strategies that students use to trace code: an analysis based in grounded theory. In *ICER '05: Proceedings of the first international workshop on Computing education research*, pages 69–80, New York, NY, USA, October 2005. ACM.

[64] J. H. Flavell. *Metacognitive Aspects of Problem Solving*, pages 231–235. Lawrence Erlbaum Associates, Inc, New Jersey, 1976.

[65] J. H. Flavell. Metacognition and cognitive monitoring: A new area of cognitive-developmental inquiry. *American Psychologist*, 34(10):906–911, October 1979.

[66] J. H. Flavell. *Cognitive Development*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[67] K. J. Ford, N. Schmitt, S. L. Schechtman, B. M. Hults, and M. L. Doherty. Process tracing methods: Contributions, problems, and neglected research questions. *Organizational Behavior and Human Decision Processes*, 43:75–117, 1989.

[68] Rob Foshay and Jamie Kirkley. *Principles for Teaching Problem Solving: Technical Paper 4*. PLATO Learning, Inc., 2003.

[69] Norman Frederiksen, Robert J. Mislevy, and Isaac I. Bejar. *Test theory for a new generation of tests.* L. Erlbaum Associates, Hillsdale, N.J., 1993.

[70] LiMin Fu. Knowledge discovery based on neural networks. *Communications of the ACM*, 42(11):47–50, 1999.

[71] U. Fuller, C. G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernán-Losada, J. Jackova, E. Lahtinen, T. L. Lewis, D. M. Thompson, C. Riedesel, and E. Thompson. Developing a computer science-specific learning taxonomy. *SIGCSE Bulletin*, 39(4):152–170, December 2007.

[72] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patters: Elements of Reusable Object Oriented Software.* Addison-Wesley Publishing Co., Inc, 1995.

[73] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[74] J. Paul Gibson and Jackie O'Kelly. Software engineering as a model of understanding for learning and problem solving. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 87–97, Seattle, WA, USA, October 2005. ACM Press.

[75] Jay N. Giedd, Jonathan Blumenthal, Neal O. Jeffries, F. X. Castellanos, Hong Liu, Alex Zijdenbos, Tomás caron Paus, Alan C. Evans, and Judith L. Rapoport. Brain development during childhood and adolescence: a longitudinal mri study. *Nature Neuroscience*, 2:861–863, 1999.

[76] Gerd Gigerenzer. *Calculated Risks: How to know when numbers deceive you.* Simon & Schuster, New York, 2002.

[77] Will D. Gillett and Eric B. Muehrcke. A pedagogical processor model. *SIGCSE Bulletin*, 15(1):159–164, February 1983.

[78] Kenneth Goldman. A concepts-first introduction to computer science. *SIGCSE 2004*, pages 432–436, March-April 2004.

[79] W. D. Gray. A view of schema theory. [review of the book: Schemas in problem solving.]. *Contemporary Psychology*, 42(2):158–159, 1997.

[80] J. G. Greeno, M. S. Riley, and R. Gelman. Conceptual competence and children's counting. *Cognitive Psychology*, 16:94–143, 1984.

[81] James Greeno. Theory and practice regarding acquired cognitive structures. *Educational Psychologist*, 10(3):117–122, 1973.

[82] Norman E. Gronlund. *Assessment of student achievement.* Allyn and Bacon, Boston ; New York, 7th edition, 2003.

[83] Mark Haas and Johnette Hassell. A proposal for a measure of program understanding. *SIGCSE '83: Proceedings of the fourteenth SIGCSE technical symposium on Computer science education*, pages 7–13, 1983.

[84] Hisham Haddad. Post-graduate assessment of cs students: Experience and position paper. *Journal for Computing Sciences in Colleges*, 18(2):189–197, December 2002.

[85] Thomas H. Haladyna. *Writing test items to evaluate higher order thinking.* Allyn and Bacon, Boston, MA, 1997.

[86] Ibrahim Abou Halloun and David Hestenes. The initial knowledge state of college physics students. *American Journal of Physics*, 53(11):1043–1048, November 1985.

[87] Jeri R. Hanly and Elliot B. Koffman. *Problem Solving and Program Design in C.* Addison Wesley, 5th edition, July 2006.

[88] J. T. Hart. Memory and the feeling-of-knowing experience1. *Journal of Educational Psychology*, 56(4):208–216, 1965.

[89] Orit Hazzan, Yael Dubinsky, Larisa Eidelman, Victoria Sakhnini, and Mariana Teif. Qualitative research in computer science education. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 408–412, New York, NY, USA, March 2006. ACM.

[90] David Hestenes, Malcolm Wells, and Gregg Swackhamer. Force concept inventory. *The Physics Teacher*, 30(3):141–158, 1992.

[91] Richards J. Heuer. *Psychology of Intelligence Analysis.* Center for the Study of Intelligence Central Intelligence Agency, http://www.cia.gov/csi/books/19104/, 1999.

[92] James Hiebert. *Conceptual and Procedural Knowledge: The Case of Mathematics.* Lawrence Erlbaum Associates, Inc, New Jersey, 1986.

[93] James Hiebert and Thomas P. Carpenter. *Learning and Teaching with Understanding*, chapter 4, pages 65–97. Macmillan, New York, 1992.

[94] Rowan W. Hollingworth and Catherine McLoughlin. Developing science students' metacognitive problem solving skills online. *Australian Journal of Educational Technology*, 17(1):50–63, 2001.

[95] Philip Holmes, Chuck Niederriter, Robert M. Panoff, and Ernest Sibert. Undergraduate computational science education (panel session). In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 419–420, New York, NY, USA, 2000. ACM Press. Chairman-Angela B. Shiflet.

[96] Mary A. Hudak and David E. Anderson. Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology*, 17(4):231–234, December 1990.

[97] Earl B. Hunt. *Concept learning, an information processing problem.* Wiley, New York, 1966.

[98] Barbel Inhelder and Jean Piaget. *The Growth of Logical Thinking from Childhood to Adolescence.* Basic Books, Inc, USA, 1958.

[99] Maria Jakovljevic. Concept mapping and appropriate instructional strategies in promoting programming skills of holistic learners. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 308–315, Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[100] Ljubomir Jerinic and Vladan Devedzic. A survey of components for intelligent tutoring pedagogical aspects of get-bits model. *SIGCUE Outlook*, 27(1):3–24, January 1999.

[101] C. G. Johnson and U. Fuller. Is bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, February 2006.

[102] P. Johnson-Laird. *How could consciousness arise from the computations of the brain?*, chapter 4, pages 101–105. Basil Blackwell, Oxford, 1987.

[103] P. N. Johnson-Laird, Vittorio Girotto, and Paolo Legrenzi. *Mental Models: a gentle guide for outsiders.* The Interdisciplinary Committee on Organizational Studies, http://www.si.umich.edu/ICOS/gentleintro.html, April 1998.

[104] Barbara Johnston. *Java Programming Today.* Prentice Hall, October 2003.

[105] Anthony E. Kelly. *Handbook of Research Design in Mathematics and Science Education.* Lawrence Erlbaum Associates, Inc, Mahwah, N.J., 2000.

[106] Malcolm S. Knowles. *The modern practice of adult education : from pedagogy to andragogy.* Follett Pub. Co., Chicago, 1980.

[107] Malcolm S. Knowles, Elwood F. III Holton, and Richard A. Swanson. *The Adult Learner: The Definitive Classic in Adult Education and Human Resource Development.* Gulf Publishing Company, Houston, Texas, USA, 5th edition, 1998.

[108] Stephen Krause, James Birk, Richard Bauer, Brooke Jenkins, and Michael J. Pavelich. Development, testing, and application of a chemistry concept inventory. *34th ASEE/IEEE Frontiers in Education Conference*, page T1G, October 2004.

[109] Barry L. Kurtz. Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. *SIGCSE Bull.*, 12(1):110–117, 1980.

[110] Hannu Kuusela and Pallab Paul. A comparison of concurrent and retrospective verbal protocol analysis. *American Journal of Pyschology*, 113(3):387–404, Autumn 2000.

[111] Imre Lakatos. *Proofs and refutations : the logic of mathematical discovery.* Cambridge University Press, Cambridge, MA, 1976.

[112] Lesley Pek Wee Land, Aybuke Aurum, and Meliha Handzic. Capturing implicit software engineering knowledge. *IEEE*, pages 108–114, 2001.

[113] A. E. Lawson. *Science Teaching and the Development of Thinking.* Wadsworth, Belmont, CA, 1995.

[114] A. E. Lawson. *Classroom Test of Scientific Reasoning: Multiple Choice Version.* Arizona State University, Tempe, AZ, revised edition edition, August 2000.

[115] A. E. Lawson. *The neurological basis of learning, development, and discovery : implications for science and mathematics instruction.* Kluwer Academic Publishers, Dordrecht; Boston, 2003.

[116] W. Leahy, G. Cooper, and J. Sweller. *Interactivity and the constraints of cognitive load theory*, chapter 7, pages 89–103. Palgrave MacMillan, New York, NY, 2003.

[117] John Lewis and Joseph Chase. *Java Software Structures: Designing and Using Data Structures.* Addison Wesley, 2nd edition, December 2004.

[118] Tracy L. Lewis and Mary Beth Rosson. A measure of design readiness: Using patterns to facilitate teaching introductory object-oriented design. *OOPSLA '02*, pages 8–9, November 2002.

[119] Loucas Louca, Andrew Elby, David Hammer, and Trisha Kagey. Epistemological resources: Applying a new epistemological framework to science instruction. *Educational Psychologist*, 39(1):57–68, 2004.

[120] George F. Luger. *Representation and Intelligence: The AI Challenge*, chapter 6-7, pages 197–302. Addison-Wesley Publishing Co., Inc, 4th edition, 2002.

[121] Howard Margolis. *Patterns, Thinking, and Cognition: A Theory of Judgment.* The University of Chicago Press, Chicago, 1987.

[122] Jane Margolis and Allan Fisher. *Unlocking the clubhouse: women in computing.* MIT Press, Cambridge, MA, 2002.

[123] David Marr. *Vision: a computational investigation into the human representation and processing of visual information.* W. H. Freeman and Company, San Francisco, 1982.

[124] S. P. Marshall. *Assessing Schema Knowledge*, chapter 7, pages 155–180. Lawrence Erlbaum Associates, Inc, Hillsdale, N.J., 1993.

[125] S. P. Marshall. *Schemas in problem solving.* Cambridge University Press, Cambridge, MA, 1995.

[126] S. P. Marshall, C. A. Pribe, and J. D. Smith. Schema knowledge structures for representing and understanding arithmetic story problems. Technical Report 86-01, Center for Research in Mathematics and Science Education, San Diego State University, San Diego, CA, March 1987.

[127] Sandra P. Marshall. *The Assessment of Schema Knowledge for Arithmetic Story Problems: A Cognitive Science Perspective*, chapter 10, pages 155–168. American Association for the Advancement of Science, Washington, 1990.

[128] Sandra P. Marshall, Barthuli, Kathryn E., Margaret A. Brewer, and Frederic E. Rose. Story problem solver: A schema based system for instruction. ONR Contract 89-01, Center for Research in Mathematics and Science Education, San Diego State University, San Diego, CA, February 1989.

[129] C. Dianne Martin. Computing curricula 2001: Reverse engineering a computer science curriculum (part 2). *inroads - The SIGCSE Bulletin*, 35(2):9–10, June 2003.

[130] Richard E. Mayer. *Thinking, Problem Solving, Cognition.* W. H. Freeman and Company, New York, 2nd edition, 1992.

[131] Richard E. Mayer. Cognitive, metacognitive and motivational aspects of problem solving. *Instructional Science*, 26(1-2):49–63, 1998.

[132] Richard E. Mayer, Jennifer L. Dyck, and William Vilberg. Learning to program and learning to think: What's the connection? *Communications of the ACM*, 29(7):605–610, 1986.

[133] Ron McClamrock. Marr's three levels: A re-evaluation. *Minds and Machines*, 1(2):185–196, May 1991.

[134] Katherine B. McKeithen, Judith S. Reitman, Henry H. Rueter, and Stephen C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

[135] Curtis McKnight, Andy Magad, Teri J. Murphy, and Michelynn McKnight. *Mathematics Education Research: A Guide for the Research Mathematician.* American Mathematical Society, 2000.

[136] Susan Mengel and William Lively. Using a neural network to predict student responses. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 669–676, New York, NY, USA, 1992. ACM Press.

[137] J. Metcalfe and A. P. Shimamura. *Metacognition: Knowing about Knowing*. MIT Press, Cambridge, MA, 1994.

[138] M. Minsky. Form and content in computer science. *J.A.C.M.*, pages 1–30, January 1972.

[139] M. Minsky. *A framework for representing knowledge*, chapter 10, pages 211–277. McGraw-Hill, New York, 1975.

[140] Robert J. Mislevy. *Foundations of a New Test Theory*, chapter 2, pages 19–39. Lawrence Erlbaum Associates, Inc, Hillsdale, N.J., 1993.

[141] M. B. Nakhleh. Students' models of matter in the context of acid-base chemistry. *Journal of Chemical Education*, 71(6):495–499, June 1994.

[142] Pavol Navrat. Hierarchies of programming concepts: Abstraction, generality, and beyond. *SIGCSE Bulletin*, 26(3):17–28, September 1994.

[143] Craig E Nelson. Student diversity requires different approaches to college teaching, even in math and science. *American Behavioral Scientist*, 40(2):165–175, November/December 1996.

[144] Craig E Nelson. *On the Persistence of Unicorns: The Trade-Off between Content and Critical Thinking*, chapter 14, pages 168–183. Pine Forge Press, Thousand Oaks, California, 1999.

[145] Craig E Nelson. Achieving higher-level educational outcomes: Fostering critical thinking & mature valuing across the curriculum. Technical report, First Annual Conference on Teaching & Learning, University of Windsor and Oakland University, 2007.

[146] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, NJ, 1972.

[147] S. Travis Nielsen and Douglas M. Campbell. Current trends in computer science graduate admissions: A survey of the top 108 programs. *SIGCSE Bulletin*, 31(2):31–34, June 1999.

[148] Ikujiro Nonaka. A dynamic theory of organizational knowledge creation. *Organizational Science*, 5(1):14–37, 1994.

[149] Ikujiro Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, 1995.

[150] Hyacinth S. Nwana. Is computer science education in crisis? *ACM Computing Surveys*, 24(4):322–324, 1997.

[151] J. O'Kelly, S. Bergin, S. Dunne, P. Gaughran, J. Ghent, and A. Mooney. Initial findings on the impact of an alternative approach to problem based learning in computer science. Technical report, PBL International Conference 2004, Pleasure by Learning, Mexico, 2004.

[152] J. O'Kelly and J. P. Gibson. Pbl: Year one analysis-interpretation and validation. Technical report, PBL International Conference 2005, PBL In Context - Bridging work and Education, Lahti, Finland, 2005.

[153] J. O'Kelly, A. Mooney, J Ghent, P. Gaughran, S. Dunne, and S. Bergin. An overview of the integration of problem based learning into an existing computer science programming module. Technical report, PBL International Conference 2004, Pleasure by Learning, Mexico, 2004.

[154] Committee on Undergraduate Science Education. *Science Teaching Reconsidered: A Handbook*. National Academy Press, Washington, 2003.

[155] Oxford Reference Online. *The Concise Oxford English Dictionary.* Oxford University Press, http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t23.e30870, 11th edition, 2006.

[156] Randall C. O'Reilly. *Levels of Analysis.* University of Colorado Boulder, http://psych-www.colorado.edu/ oreilly/cecn/node11.html, April 2000.

[157] Jason W. Osborne. Measuring metacognition: Validation of the assessment of cognition monitoring effectiveness. *Dissertation Abstracts International Section A: Humanities and Social Sciences*, 59(5-A):1459, November 1998.

[158] Kaori Ozawa. *Introduction to Cognitive Science: Chapter 2 - Marr.* www.wam.umd.edu/ kozawa/NACS728/NACS728_Marr.doc, December 2004.

[159] Richard Pak. Recap of our meeting on tuesday... private communication, October 2007.

[160] Jennifer Parham, Donald Chinn, and D. E. Stevenson. Using blooms taxonomy to code verbal protocols of students solving a data structure problem. In *Proceedings of the ACM Southeast Conference: Clemson, SC 2009*, March 2009.

[161] Jennifer R. Parham. An assessment and evaluation of computer science education. *Journal of Computing in Small Colleges*, 19(2):115–127, December 2003.

[162] Michael Pavelich, Brooke Jenkins, James Birk, Richard Bauer, and Steve Krause, editors. *Development of a Chemistry Concept Inventory for Use in Chemistry, Materials and other Engineering Courses.* 2004 American Society for Engineering Education Annual Conference & Exposition, American Society for Engineering Education, 2004.

[163] J. Piaget. *The Psychology of Intelligence.* Routledge & Paul, London, 1967.

[164] J. Piaget. *Structuralism.* Basic Books, Inc, New York, 1970.

[165] J. Piaget. *Success and Understanding.* Harvard University Press., Cambridge, MA, 1978.

[166] G. Polya. *How to Solve It: A New Aspect of Mathematical Method.* Princeton University Press, Princeton, NJ, 2nd edition, 1957.

[167] G. Polya. *Mathematics and plausible reasoning: Patterns of plausible inference*, volume 2. Princeton University Press, Princeton, NJ, 1968.

[168] R. Porter and P. Calder. Patterns in learning to program: an experiment? In R. Lister and A. Young, editors, *In Proceedings of the Sixth Conference on Australasian Computing Education - Volume 30*, pages 241–246, Dunedin, New Zealand, 2004. ACM International Conference Proceeding Series, vol. 57, Australian Computer Society.

[169] Michael Pressley and Peter Afflerbach. *Verbal protocols of reading: the nature of constructively responsive reading.* Lawrence Erlbaum Associates, Inc, Hillsdale, N.J., 1995.

[170] Edward A. Price and Marcy P. Driscoll. An inquiry into the spontaneous transfer of problem-solving skill. *Contemporary Educational Psychology*, 22:472–494, 1997.

[171] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. Self-efficacy and mental models in learning to program. *ITICSE 2004*, pages 171–175, June 2004.

[172] Cecylia Rauszer. *Algebraic methods in logic and in computer science.* Institute of Mathematics, Polish Academy of Sciences, Warszawa, 1993.

[173] Daniel Reisberg. *COGNITION: Exploring the Science of the Mind*. W. W. Norton & Company, New York, 1997.

[174] M. Reiss, M. Behr, R. Lesh, and T. Post. Cognitive processes and products in proportional reasoning. *Proceedings of the Ninth International Conference for the Psychology of Mathematics Education*, pages 352–356, July 1985.

[175] Dawn Rickey and Angelica M. Stacy. The role of metacognition in learning chemisty. *Journal of Chemical Education*, 77(7):915–920, July 2000.

[176] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.

[177] Kenneth H. Rosen. *Common Mistakes in Discrete Mathematics*, pages 406–414. McGraw-Hill Higher Education, 6th edition, 2006.

[178] D. E. Rumelhart. *Schemata: The building blocks of cognition.*, chapter 2, pages 33–58. Lawrence Erlbaum Associates, Inc, Hillsdale, N.J., 1980.

[179] David E. Rumelhart and James L. McClelland. Levels indeed! a response to broadbent. *Journal of Experimental Psychology*, 114(2):193–197, 1985.

[180] David E. Rumelhart and Donald A. Norman. *Analogical Processes in Learning*, chapter 11, pages 335–360. Lawrence Erlbaum Associates, Inc, Hillsdale, N.J., 1981.

[181] Gilbert Ryle. *The Concept of Mind*. Barnes & Noble, 1962.

[182] Penelope Sanderson and Carolanne Fisher. Exploratory sequential data analysis: Foundations. *Human-Computer Interaction*, 9:251–317, 1994.

[183] Walter Schaeken, Andre Vandierendonck, Walter Schroyens, and Gery d'Yewalle. *The Mental Models Theory of Reasoning: Refinements and Extensions*. Lawrence Erlbaum Associates, Inc, Mahwah, N.J., 2007.

[184] R. C. Schank and R. Abelson. *Scripts, Plans, Goals, and Understanding*. Erlbaum Assoc., Hillsdale, N.J., 1977.

[185] I. Scheffler. *Conditions of Knowledge: An introduction to epistemology and education*. University of Chicago Press, Chicago, 1965.

[186] A. H. Schoenfeld. *Learning to think mathematically: Problem solving, metacognition and sense making in mathematics*, pages 334–370. Macmillan, New York, 1992.

[187] Bennett L. Schwartz and Timothy J. Perfect. *Introduction: toward an applied metacognition*, chapter 1, pages 1–14. Cambridge University Press, 2002.

[188] Elaine Seymour and Nancy M. Hewitt. *Talking about leaving : why undergraduates leave the sciences*. Westview Press, Boulder, Colorado, 1997.

[189] Russell Shackelford, James H. Cross II, Gordon Davies, John Impagliazzo, Reza Kamali, Richard LeBlanc, Barry Lunt, Andrew McGettrick, Robert Sloan, and Heikki Topi. *Computing Curricula 2005: the overview report*. The Association for Computing Machinery, The Association for Information Systems, and The Computer Society, September 2005.

[190] Stewart Shapiro. *Foundations without Foundationalism: A Case for Second-order Logic*. Clarendon Press, Oxford, 1991.

[191] Michael Shayer and Philip Adey. *Towards a Science of Science Teaching: Cognitive development and curriculum demand*. Richard Clay Ltd, London, 1981.

[192] Angela B. Shiflet. Computer science with the sciences: An emphasis in computational science. *inroads - The SIGCSE Bulletin*, 34(4):40–43, December 2002.

[193] B. F. Skinner. The shame of american education. *American Psychologist*, 39(9):947–954, September 1984.

[194] John F. Sowa. *Schemata*, chapter 2.4, pages 42–51. Addison-Wesley Publishing Co., Inc, 1983.

[195] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley Publishing Co., Inc, 1984.

[196] Robert J. Sternberg. *Beyond IQ : a triarchic theory of human intelligence*. Cambridge University Press, Cambridge, MA, 1985.

[197] Robert J. Sternberg. *Intelligence Applied: Understanding and Increasing Your Intellectual Skills*. Harcourt Brace Jovanovich, Inc., 1986.

[198] Robert J. Sternberg. *The Triarchic Mind: A New Theory of Human Intellegence*. Penguin Books USA Inc., New York, NY, 1989.

[199] Robert J. Sternberg. *Thinking and Problem Solving*. Academic Press., San Diego, 1994.

[200] Robert J. Sternberg and Janet E. Davidson. *The psychology of problem solving*. Cambridge University Press, Cambridge, UK; New York, 2003.

[201] Robert J. Sternberg and Peter A. Frensch. *Complex problem solving: principles and mechanisms*. L. Erlbaum Associates, Hillsdale, N.J., 1991.

[202] Robert J. Sternberg and Elena L. Grigorenko. *The psychology of abilities, competencies, and expertise*. Cambridge University Press, Cambridge, UK; New York, 2003.

[203] Robert J. Sternberg and Joseph A. Horvath. *Tacit knowledge in professional practice: researcher and practitioner perspectives*. L. Erlbaum Associates, Mahwah, N.J., 1999.

[204] Robert J. Sternberg and Edward E. Smith. *The Psychology of human thought*. Cambridge University Press, Cambridge; New York, 1988.

[205] Matthias Steup. *The Analysis of Knowledge*. Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/knowledge-analysis/, January 2006.

[206] D. Stevenson and J. Parham. Problem based and case based methods in computer science. *Creative College Teaching Journal*, 2006.

[207] D. E. Stevenson. computer science as problem solving discipline. private communication, January 2008.

[208] S. Stevenson. Problem solving principles and free-writing techniques in a computer science class. In *ACMSE'03, 41st ACM Southeast Regional Conference*, Savannah, Georgia, March 2003.

[209] Angel Syang and Nell B. Dale. Computerized adaptive testing in computer science: assessing student programming abilities. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, pages 53–56, New York, NY, USA, 1993. ACM Press.

[210] J. Tartar and J. P. Penny. Undergraduate education in computing science some immediate problems. In *SIGCSE '72: Proceedings of the second SIGCSE technical symposium on Education in computer science*, pages 1–7, New York, NY, USA, 1972. ACM Press.

[211] Esther Thelen and Linda B. Smith. *A Dynamic Systems Approach to the Development of Cognition and Action*. MIT Press, Cambridge, MA, 1994.

[212] Kenneth Tobin and William Capie. Development and validation of a group test of logical thinking. *Educational and Psychological Measurement*, 41(2):4l3–424, 1981.

[213] Kenneth Tobin and William Capie. The test of logical thinking. Technical report, Journal of Science and Mathematics Education in S.E. Asia., 1984.

[214] David E. Trowbridge and Lillian C. McDermott. Investigation of student understanding of the concept of velocity in one dimension. *American Journal of Physics*, 48(12), December 1980.

[215] David E. Trowbridge and Lillian C. McDermott. Investigation of student understanding of the concept of acceleration in one dimension. *American Journal of Physics*, 49(3), March 1981.

[216] Philip E. Vernon. *The measurement of abilities*. University of London Press, London, 1972.

[217] L. S. Vygotsky. *Mind in society: the development of higher psychological processes*. Harvard University Press, Cambridge, MA, 1978.

[218] L. S. Vygotsky. *The Genesis of Higher Mental Functions*, pages 158–170. M. E. Sharpe, Armonk, NY, October 1981.

[219] Lev Vygotsky. *Thought and language*. MIT Press, Cambridge, MA, 1986.

[220] R. T. White. chapter 2, pages 70–75. Oxford, Pergamon, 1997.

[221] Wikipedia. *Immanuel Kant*. Wikimedia Foundation, Inc., http://en.wikipedia.org/wiki/Immanuel_Kant, August 2007.

[222] Wikipedia. *Schema (Kant)*. Wikimedia Foundation, Inc., http://en.wikipedia.org/wiki/Schema_%28Kant%29, June 2007.

[223] Wikipedia. *Schema (Psychology)*. Wikimedia Foundation, Inc., http://en.wikipedia.org/wiki/Schema_%28psychology%29, July 2007.

[224] Stanley Wileman, John Konvalina, and Larry J. Stephens. Factors influencing success in beginning computer science courses. *Journal of Educational Research*, 74(4):223–226, March-April 1981.

[225] Cecile Yehezkel, Mordechai Ben-Ari, and Tommy Dreyfus. Computer architecture and mental models. *SIGCSE 2005*, pages 101–105, 2005.

[226] Caroline E. Zsambok, Lee Roy Beach, and Gary Klein. *A LITERATURE REVIEW OF ANALYTICAL AND NATURALISTIC DECISION MAKING*. Klein Associates Inc., December 1992.