NATURAL LANGUAGE TUTORING AND THE NOVICE PROGRAMMER

by

H. Chad Lane

B.S. Mathematics & Computer Science, Truman State University, 1995

M.S. Computer Sciences, University of Wisconsin-Madison, 1997

Submitted to the Graduate Faculty of
the Department of Computer Science in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2004

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

H. Chad Lane

It was defended on

8 September 2004

and approved by

Dr. Kurt VanLehn, Depts. of Computer Science & Psychology

Dr. Diane Litman, Dept. of Computer Science

Dr. Janyce Wiebe, Dept. of Computer Science

Dr. Peter Brusilovsky, Dept. of Information Science

Dr. Marian Petre, Dept. of Computing, The Open University, UK

Dissertation Director: Dr. Kurt VanLehn, Depts. of Computer Science & Psychology

NATURAL LANGUAGE TUTORING AND THE NOVICE PROGRAMMER

H. Chad Lane, PhD

University of Pittsburgh, 2004

For beginning programmers, inadequate problem solving and planning skills are among the most salient of their weaknesses. As a result, they often struggle profoundly when confronted with a programming task. One reason for this is that novices, by definition, lack much of the tacit knowledge that underlies effective programming. This dissertation examines the efficacy of natural language tutoring (NLT) to foster acquisition of this tacit knowledge. The hypothesis is that NLT is superior to reading alone.

Coached Program Planning (CPP) is proposed as a solution to the problem of teaching the tacit knowledge of programming. The general aim is to cultivate the development of such knowledge by eliciting and scaffolding the problem solving and planning activities that novices are known to underestimate or bypass altogether. The specific goals of CPP are to elicit goal decompositions and program plans from students in natural language. A variety of tutoring tactics are used that leverage students' intuitive understandings of the problem, how it might be solved, and the underlying concepts of programming. ProPl("pro-PELL"), a dialogue-based intelligent tutoring system based on CPP, is also described. ProPl is intended to help students plan their programs before they write them. Keyword and phrase spotting is used to understand student input and dialogue management can loosely be classified as using a finite-state model. Knowledge Construction Dialogues (pre-authored hierarchical structures) are used to implement a variety of the CPP tutoring tactics, including many for eliciting goals and fleshing out plan details.

In an evaluation, the primary findings were that students who received tutoring from ProPl seemed to exhibit an improved ability compose plans and displayed behaviors suggestive of thinking at greater levels of abstraction than students in a read-only control group. ProPl students were more successful assembling algorithms and worked with code at the level of plans rather than in the usual, line-by-line approach that novices tend to adopt. Surprisingly, no differences were detected on written design questions. The major finding is therefore that NLT appears to be effective in teaching program composition skills, but less so at raising the competency of students to use natural language.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

This is my first preface. Let me tell you something: I've been waiting a long time to write this. Of all the dissertations I've cracked open over the years as a graduate student, it was (and still is) my favorite part. I believe it is a tremendous feeling to think about the reasons we do what we do and to consider those involved in our lives.

The best place to start in the thanking frenzy has to be with all of my former students. I was lucky enough to have taught beginning programming throughout my graduate student years, first at the University of Wisconsin-Madison and then again at the University of Pittsburgh. Early on, I had no idea what was happening. I was repeatedly fascinated by what these students were experiencing. The difficulties they faced and the ensuing (often intense) frustrations were as compelling for me as anything I ever experienced. I looked at countless programs that *should never have been written.* The raging battles they fought with the compiler seemed, to me, to be monumental wastes of time. I wanted students to plan ahead! For one semester, I required that students turn in pseudocode a week before their assignment was due. Their attempts were below poor – some students went ahead and wrote the whole program, while others copied a different program from the book. It became clear to me at that point: students were not ready for programming. They needed more than just a problem statement and a compiler.

Luckily, around January of 2001, my adviser, Dr. Kurt VanLehn, handed me a list with about 10 or 12 references to papers about novice programmers. He told me that it was time we started putting some of this inspiration from the classroom into my research and ultimately, turn it into a dissertation. That was a great day, and the end result is in your hands now. I cannot thank Kurt enough for all of the sagacious guidance, unwavering support and encouragement, and the friendship he has extended to me over the years. I could not have had a better adviser.

There are also many people in the Computer Science department who have helped me get done what needed to be done: Kathy O'Connor, Loretta Shabatura, Nancy Kreuzer, Don Bonidie, Dr. George Novacky, Dr. Panos Chrysanthis, and Dr. Rami Melhem. I've also been extremely lucky to have had nothing but wonderful people in LRDC to work with: Eric Fussenegger, Jo-Anne Krevy, De Ivanhoe, Tamara Cathcart, and everyone else on the 2nd and 5th floors.

On the research side of the coin, I have also been very lucky to have had brilliant fellow graduate students and highly skilled friends. Kurt and Dr. Micki Chi deserve a lot of credit for assembling and maintaining such strong research environments. Quite a lot of people have helped with my project in particular: Dr. Pam Jordan (who *volunteered* to read my boring proposal), Collin Lynch (who *volunteered* to help run my subjects when I had to be out of town), Dumiszewe Bhembi (who did the same thing as Collin), Linwood Taylor (a masterful programmer who can make Lisp and Java talk to each other like they were old beer drinking buddies from college), Bob Hausmann (who taught me everything I know about experimental design and SAS), and Mark Fenner (who helped code some data and provided some great conversations). I also owe a lot to Kurt's other graduate students: Noboru Matsuda, Min Chi, Mike Ringenberg, Chas Murray, and the brand new shiny Dr. Stephanie Siler. Our weekly meetings were always intriguing and useful, even though we never could get the order straight. I have also have been fortunate to have had fantastic professors while at Pitt who helped lay the groundwork that enabled me to write a dissertation and prepare for the rest of my career: Dr. Bruce Buchanan, Dr. Gaea Leinhart, Dr. Kevin Crowley, and Dr. Diane Litman, who is also on my committee. I would also like to thank the rest of my committee for taking the time out of their busy lives to help me do this and for always being so willing to negotiate times, places, etc. My other committee members are Dr. Jan Wiebe, Dr. Peter Brusilovsky, and Dr. Marian Petre from the Open University in Milton Keynes, UK.

Finally, and most importantly, without my family, I most definitely would not be writing my first preface. There has not been a minute in my 31 years of life when they have not been there to support, love, and provide. Bart, my brother, has kept me keen to what is funny about the world and my spirits high; Melissa, my sister, has been a constant source of wisdom and calmness for me; for my Mom, I would have to write a dissertation-sized preface to list what she has done for me over the years – the cards, food, clothes, etc. etc – the love and support is never-ending from her, and I know how lucky I am; my Dad inspired me to love to teach and to love to learn – insights from this 40+ years of teaching have directly affected the content of this dissertation, and on top of all that, I love golf because of him. I would also like to thank Nichole for her willingness to juggle schedules and for her friendship over the years. Finally, I am dedicating this dissertation and the effort that went into it to my son, Rowan. Even though the world awaits his first complete sentence, he has said more to me over the last 2 years than anyone.

HCL, September 2004

1.0 INTRODUCTION

A computer program is more than text on a screen. A completed program by itself reveals very little about the process that went into creating it or the knowledge required to produce it. In addition to knowing a particular programming language, creating a program requires problem solving, design, editing, and debugging skills. There is also an undeniable level of creativity required to program well. Teaching these skills and the requisite knowledge is turning out to be one of the more challenging educational problems of our time. Novice programmers represent a compelling group of students to study because of the habits they adopt and beliefs they form. This dissertation presents a new approach to tackling this now classic problem.

The goal of this work is to explore an application of artificial intelligence in an effort to accelerate the development of programming skills in beginners. Although the techniques of AI have been used many times over in novice programming systems, this dissertation demonstrates how natural language technology can be used to scaffold the earliest stages of writing a program. The culmination of the work is a dialogue-based intelligent tutoring system for novice program design that is evaluated with real students. The experiment revealed that students using the dialogue-based tutor improved in several key ways over students who read the same material. Most importantly, tutored students demonstrated an improved ability to produce conceptually distinct, but textually integrated program code. The purpose of this opening chapter is to define the problem being addressed, explain why it is important and worthwhile to attack it, and preview the proposed solution and remainder of the dissertation.

1.1 MOTIVATION

With the increasing prominence of computers in the everyday lives of people, there has been a concomitant increase in the number of people interested in acquiring programming skills. Universities are increasingly offering an introductory programming course as an elective that satisfies core degree requirements. This has led to an increase in the number of "non-majors" enrolled in such courses and thereby broadened the traditional notion of who novice programmers are. To meet the needs of this new class of beginner (i.e., those not bound for computing careers), some departments have redesigned their introductory courses (Forte and Guzdial 2004; Herrmann et al. 2003), while many others have begun offering courses for non-majors, designed for the true beginner. For many of these students, this will likely be their only exposure to the concepts of computation and algorithm design. Their experiences, good and bad, will act as the backdrop for all of the interactions they have with programmers and computer scientists for the rest of their lives. It is important for educators and researchers to recognize the importance of this group, and to take advantage of this brief window of opportunity in these students' lives.

This dissertation is not only motivated by non-majors, however: beginning programmers in general struggle with many of the same issues. Difficulties that stem from inadequate problem-solving abilities are present in all classes of students, including those majoring in computer science. It is also hard to teach programming: evaluations of programming skill in introductory courses done in the 80's (Soloway et al. 1982) and again in the late 90's (McCracken et al. 2001) revealed an alarming lack of competence in students, even after a full year of programming instruction. Clearly, the search for effective methods of teaching programming is far from over. Strangely, many of the methods used to teach programming are partly responsible for the problems novices face. For example, a class syllabus built around programming constructs sends the message that programming is mostly *about those constructs*. Rather than reading a problem and *then* determining how to solve it with the tools of a language, novices learn to pick a tool (e.g., a for statement) and then seek to fit it to the problem. There is certainly no universal solution to the problem of teaching programming; however, it is precisely this challenge, the difficulties novices face, and the misconceptions that underly them that make novice programmers a compelling subject for research.

1.2 THE PROBLEM AND APPROACH

Novices experience a great deal of frustration and difficulty when programming (du Boulay 1989; Pea and Kurland 1983; Pintrich, Berger, and Stemmer 1987; Robins, Rountree, and Rountree 2003). Although there are many causes for their troubles, a large portion can be directly traced to poor planning and design skills. To understand the cognitive skills and knowledge that underlie programming, numerous researchers have studied how experienced programmers generate, comprehend, and maintain computer programs (Linn 1985; Rich 1981; Rist 1995; Soloway and Ehrlich 1984). The resulting theories tend to assert the existence of reusable "chunks" of knowledge representing solution patterns that achieve different kinds of goals. In this dissertation, these chunks are referred to as *schemata*. When a programming goal is recognized, a programmer must retrieve or create a schema to achieve that goal. For example, if the goal is to determine the length of an arbitrarily long sequence of numbers, the programmer would use a counter-schema. It consists of four components: an initialization step, an enclosing loop, an increment step, and a print step after the loop (assuming a requirement exists to output the length).

In what seems like second nature, experienced programmers are able to implement a counter with little or no difficulty. The knowledge has been been internalized, generalized, and stored. Novices are at a disadvantage, however, because they have not yet built up the abstractions. All programmers must go through this acquisition process at some point, and for many, it is quite a challenge. In addition, complicating the problem for novices is the observation that, at some points during learning, they are forced to construct "stories" that explain observed events to themselves, which are often inconsistent and/or incomplete (Taylor 1999). This suggests that it is dangerous to leave novices alone during the tenuous time of learning to program. *The problem being addressed in this dissertation is therefore how to scaffold the development of the tacit knowledge of programming in novices.* I seek to uncover, implement, and evaluate a tutoring model that reveals and makes this knowledge more accessible.

The basic approach is to use natural language dialogue to draw on students' commonsense understandings of programming and the problems being solved. It is often necessary to help them transition from their initial understandings to a more algorithmic view of the solution. Such a situation appears in figure 1.1. This is a dialogue between the author (playing the role of tutor) and student over a network. The tutor[1] is helping the student understand how a sequence of values could be counted in a program. The goal is simply to create a sequence of numbers according to rules that are given, and display how many items were generated by the loop.

Each line in this dialogue (and all subsequent dialogues) is broken down by utterances. In other words, each line number corresponds to one, typed contribution to the dialogue. The sub-dialogue begins with a question to the student asking if s/he has any ideas about how to implement a counter. The student is unsure how to go about it, but does take the initiative and provides a shallow response (lines 2-3). The tutor ignores this answer and proceeds to ask how s/he went about it in an earlier example. The response, "I just counted them by hand" suggests a *post-sequence* counting strategy, meaning that the student looked back over the sequence after it was created to determine the length.

The commonsense approach to counting is expected in novices. The student would have no reason to adopt a more dynamic counting strategy when doing a sequence by hand. The problem is that this intuitive strategy does not translate well into an algorithmic solution. It might conjure up strange ideas, such as reading from the screen or storing all of the values in a variable to count them later (arrays are typically taught after loops). The point of asking, however, is to help the student recognize this crucial difference. The dialogue continues in figure 1.2, which shows the tutor gently nudging the student towards a more algorithmic view of counting. The tactic is to ask the student to "watch" the count as successive values of the sequence are produced for a particular example (lines 11-16). Although the tutor draws the conclusion for the student in this case (line 17), the student does provide some

[1] To reduce confusion between my roles as researcher and as tutor in the study, I will use "tutor" to refer to myself in the tutoring sessions.

1 T Can you tell me how we can use count up the items in the sequence?
2 S **I'm not sure**
3 S **make a procedure to do it**
4 T So tell me when you said there were 7 items in our example earlier,
5 T how did you know?
6 S **I just counted them by hand**
7 T Unfortunately our program cannot do it that way, at least not easily.

Figure 1.1: The student describes a commonsense method of counting items in a sequence. This causes a problem for novice programmers because they naturally attempt to implement a similar strategy in their programs.

8 T So can you think of a different way to count? A way that does not
9 T require you to look back over your work?
10 *(pause)*
11 T It's ok... let's go back to the example.
12 T If we start at 6, how many items is that so far?
13 S **1**
14 T Good. You said earlier that 3 is next. How many items now?
15 S **2**
16 T After that is 10, takes us up to 3. And so on.
17 T So rather than counting them up when we're done, we keep track of it
18 T as we go along.
19 S **ohhh!**
20 S **do we need to make another variable then?**

Figure 1.2: The tutor uses an example to teach the student how to count algorithmically; i.e., by incrementing a counter variable.

positive confirmation (lines 19), even suggesting that a new variable would be needed in the program (line 20).

As mentioned earlier, novices are essentially breaking ground on their mental library of programming schemata. In the example, the tutor is helping the student develop an algorithmic approach to counting a sequence. The ensuing dialogue included a discussion of initializing the counter variable, placing the increment step inside the loop, and printing the result after the loop. In other words, the tutor is scaffolding the creation of the counter-schema. When teachers send students off to write their programs, this essentially leaves it up to each student to build these abstractions and mental libraries up for themselves. The assumption in this research is that this is a bad idea: novices are not ready to fly on their own when the extent of their background is textbook examples, labs, and lectures. There is simply too much going on for novices to manage it all *and* learn effectively from the experience. The evidence for this conclusion comes simply from the massive number of papers for almost three decades that report the many difficulties of novice programmers (many of which are cited in the bibliography as well as in Robins, Rountree, and Rountree 2003).

The use of more open-ended questions, such as the one posed in figure 1.1 (line 1), is an important component of the overall tutoring strategy proposed by this dissertation. Novices do not often know what questions to ask themselves, and so posing questions to them allows the important problem solving issues to be addressed without the distraction of the programming language, compiler, etc. It also forces students to search the space of design choices and explore different alternatives prior to engaging in an implementation. On top of this, it is also safer to make a suggestion in the presence of the tutor because misunderstandings of the problem and "bad" ideas can be detected and remediated to prevent them from propagating through to the implementation phase. The tutoring model can be classified under the general notion of *pre-practice tutoring*, since tutoring occurs before actual problem solving with a compiler. Tutoring is intended to cover projects the student will be attempting independently at a later time, and so the hope is that their difficulties will be reduced by having been tutored.

Of course, the cost of using human tutors to tutor all students on each project in a class would be prohibitive. For this reason, ProPl ("pro-PELL", short for PROgram PLanner), a dialogue-based intelligent tutoring system, was created to provide such intervention automatically. Briefly, ProPl is web-based software that students can use prior to beginning their programming projects to help them engage in meaningful planning activities. ProPl is a partial implementation of *Coached Program Planning* (CPP), the tutoring model introduced by this dissertation.

1.3 RESEARCH GOALS AND QUESTIONS

It is generally accepted that it takes 10 years to bring a beginning programmer up to an expert level (Winslow 1996). When looked at in this light, it is hard to imagine that any intervention at all could have a perceptible impact on students given only 16 weeks to work. However, the goal here is not to produce expert programmers, but rather to help novices along during a delicate and volatile time in their acquisition of programming skills. Also, as many students in beginning programming courses will not continue in the field, a second reason to pursue this line of work is to improve their experiences, and hopefully also their attitudes of programming.

So, one top-level goal of this research is to determine if there are pedagogical benefits to providing pre-practice tutoring in programming, and if so, what are they? How do tutored students compare to students who receive no such tutoring and are left to develop programming knowledge on their own? The second goal of the research is to assess the efficacy of natural language tutoring to foster the development of programming skills and understanding of the tacit knowledge of programming in novices. The hypothesis is that dialogue-based tutoring will produce better learning than reading the same content. If so, why is dialogue-based tutoring superior to reading? Do students actually develop a heightened sense of the knowledge underlying completed programs? Does it affect their view of the tasks of programming? Are they able to use natural language to produce abstractions and decompose problems more effectively?

Research on the tutoring of novice programmers has almost exclusively focused on programming during the implementation phase. This body of work has explored intelligent support for writing functions (Anderson 1990), automatic debugging (Anderson et al. 1995), and analyses of human tutors in introductory programming courses (Littman, Soloway, and Pinto 1990). Very little work involving the tutoring of novices *before* they enter into an implementation phase has been published. There is evidence that providing conceptual guidance and helping novices establish a better context for problem solving before they begin working on a problem can have a positive effects for students learning physics (Dufresne et al. 1992), and so this research represents a similiar investigation into novice programming.

This dissertations builds on previous research on novice programming, intelligent tutoring systems, and educational dialogue systems. It is also novel in several ways:

- It presents a dialogue-based tutoring system for novice program *planning*. Several dialogue-based exist that support debugging and code improvement (discussed in section 2.3), but none work with the student in the beginning of the process, when their ideas and impressions are first being formed about the problem.

- It acts as a test of the generality of the Knowledge Construction Dialogue (KCD) technology since it is being applied to a design domain (programming) for the first time.

- The tacit knowledge of programming is being taught without introducing any concrete representation of it. Most previous approaches based on theories of schema-like programming knowledge *explicitly* identify these chunks.

- A new method of evaluating student success, namely *intention-based scoring*, is introduced to deal with the problem of assessing the *process* of programming.

In sum, it is novel to both tutor only the planning process and to do so with natural language tutoring. Why tutor only before programming? It could certainly turn out that support in all phases is best; but, to understand the impact of tutoring during each phase, I have chosen to start at the beginning. To understand the relative importance of tutoring at each phase, tutoring only in the planning phase, and observing the results, can begin to answer this question. In addition, this research is a direct result of personal teaching experiences with novice programmers. In many different situations, it was clear to me as a teacher that students were not starting "on the right track," but instead were making serious mistakes early, and paying for throughout an entire implementation. My belief was that they simply needed help establishing the context and starting place for solving programming problems. Therefore, this research is an attempt to provide an intelligent tool for pre-planning assistance and, in the best of cases, make life slightly better for beginning programmers.

1.4 OVERVIEW

The remainder of the dissertation is organized as follows. Chapter 2 provides a task analysis of programming and elaborates on the "tacit knowledge" that underlies it. The decomposition and composition problems are introduced, both of which pose difficulty for novices. A literature review on novice programmers is also presented, describing what is hard about programming, what novices do to make it even harder, and why it is a bad idea to turn them loose with a compiler.

Chapter 3 summarizes past and current research on human tutoring and intelligent tutoring systems. Human tutoring is described, why it is believed to be the best known form of teaching, and also what kinds of student behaviors are known to correlate with learning. Because the interest is specifically in automated natural language tutoring, this chapter also includes a review of existing dialogue-based educational systems.

Chapter 4 is in many ways the centerpiece of the dissertation. It sits at the confluence of the previous two chapters by describing how natural language tutoring can be applied to the problem of teaching programming. The suitability of the natural language approach is discussed, showing that it is easier to connect to students' existing knowledge through dialogue. This chapter also presents CPP, the model of planning embedded in it and the tutoring tactics used for remediation. The model is based on a three-step pattern that prescribes the identification of programming goals, elaboration on how to achieve them, and implementation of the identified techniques in pseudocode. A variety of tutoring tactics are also identified, such as the one shown in figures 1.1 and 1.2.

The topic of chapter 5 is about how students describe goals and how to achieve them, based on a corpus of program planning dialogues. This was preliminary work for building ProPl and reveals that students, in general, are able to find productive contributions to make when asked what and how questions. However, a great deal of tutorial effort is often required to refine their answers. Much of the work presented in this chapter was directly encoded into the dialogue knowledge sources used by ProPl.

Chapter 6 describes the intelligent tutoring system ProPl that implements of the tutoring model presented in the previous chapter. It applies many of the tactics identified in chapter 4. In addition, ProPl-C, a non-dialogue-based version of the system, is described. The interface and implementation of both systems are described in detail along with several sample interactions.

Chapter 7 describes two controlled evaluations: one from the human-human study of the tutoring of novice program design and the second, a controlled evaluation of ProPl. The evaluation reveals that students who used ProPl were more effective at solving the composition problem and demonstrated a preference for more abstract representations of programming knowledge. No differences in generating natural language solutions or in solving the decomposition problem were detected.

The dissertation concludes with chapter 8, which contains a summary of the general argument of the dissertation, discussion of the main findings, a review of the contributions, and discussion of future work.

2.0 LEARNING TO PROGRAM

Becoming an effective programmer requires a unique blend of technical, strategic, and practical skills – it is difficult to imagine a greater learning challenge for a beginner. This chapter begins by examining what programming entails and discusses the acquisition of programming knowledge by beginners. Motivating answers to the classic question "What makes programming hard?" are discussed, and the chapter concludes with a review of existing sytems for novice programmers.

2.1 PROGRAMMING TASK ANALYSIS

2.1.1 The tacit knowledge of programming

As discussed in chapter 1, writing a program requires skills and knowledge that go well beyond that of a particular programming language. In addition, there is a great deal of implicit knowledge behind a completed program. Studies that focus on the content and structure of this knowledge generally identify structured "chunks" of knowledge that achieve a variety of goals; however, some terminological confusions have arisen. Rich (1981) refers to such a chunk as a *programming cliche*, Linn (1985) uses *template*, Soloway (1984) calls it a *plan*, and Rist (1995) uses *schema*. More recently, the term *design pattern* has been adopted to represent similar knowledge that underlies object-oriented programs (Gamma et al. 1995). The term *schema* is used here to refer to the general form of these chunks, while *plan* is used to refer to instantiated versions of the same knowledge. That is, *plan* is used to be specific to a problem and *schema* to be generic, with placeholders where program variables sit in plans.

This distinction will be useful in later chapters when the issue of how these representations are used to drive dialogue is discussed.

2.1.2 Cognitive subtasks involved in programming

Programming is fundamentally a problem solving task. Hence, models of programming have their roots in models of general problem solving, such as those originally proposed by Polya (1957) and Newell and Simon (1972). Programming is also a *design* task (Greeno and Simon 1988). The product of programming is an artifact, not the output produced by programs, as many novices tend to believe. It is this aspect of programming that distinguishes it from problem solving in other domains such as calculus or physics. Numerous researchers have extended and recast these general models for programming (Feddon and Charness 1999; Pea and Kurland 1983; Pennington and Grabowski 1990), all of which essentially posit four necessary stages novices and experts must go through to create a computer program:

1. **understand the problem:** develop a mental representation of the task
2. **plan/design a solution:** produce a description (mental or on paper) of the program
3. **implement the solution:** translate design into program code
4. **debug program:** observe program execution and repair if necessary

Some models include *maintenance* either separately or together with debugging, which refers to tasks such as adding functionality to the program or optimizations based on runtime data. Most beginning programming classes do not address maintenance, optimization, and other tasks associated with fully-written programs to any significant degree, and this will not be addressed by this dissertation.

This is an idealized model, however. Pennington (1990, p. 47) points out four reasons why the distinction between the stages is not as clean as it may appear. First, expert programmers rarely complete one stage before beginning another – they are able to interweave work on various stages at once. Second, design takes place at different levels of abstraction; that is, different abstractions may be appropriate for different stages. Third, there may be interactions between the problem solving domain and the algorithm being developed. For example, domain-specific knowledge may be required when planning an algorithm. Fourth, the environments used by programmers may convolute the programming process. Some encourage no distinctions between the stages while others impose tight restrictions to maintain the distinctions between stages. The practices of experienced programmers do not always translate into an appropriate model for beginners, however. Although experts are able to juggle the stages of programming, and do so effectively, it is argued here that this is something novice programmers should not be allowed to attempt. They simply lack the skill and experience of experts to handle it. This issue is addressed in greater detail in section 2.2.

2.1.3 The decomposition and composition problems

In terms of a schema-based theory of programming knowledge, it is possible to elaborate on what the planning and implementation phases entail. To write a program, several studies have identified two key problems to solve (Jeffries et al. 1981; Guzdial et al. 1998):

- **Decomposition problem:** identifying the goals and corresponding schemata needed to solve the problem.
- **Composition problem:** assembling the corresponding plans and coordinating their interactions such that the problem is solved correctly.

Soloway et. al. (1988) state that the decomposition problem asks the programmer to "lay the components of the solution on the table" (p. 142). For introductory problems, goals can either be explicitly stated in the problem statement or be implicit to it. When solving the decomposition problem, the programmer must identify these goals and retrieve appropriate schemata. Although the plans that achieve goals are generally easy to understand in isolation, subtle interactions and complications can arise when multiple plans are to be merged. For example, when two plans each call for a loop, the programmer must determine if one loop in the proposed solution can satisfy both plans from the decomposition. These are not always easy decisions given that, in some cases several solution paths are acceptable, while, in others, a poor design choice made early has the potential to propagate throughout an implementation, limiting or even possibly eliminating other, higher quality paths to a solution.

Unlike the strict syntactic rules imposed by compilers, there are no steadfast rules describing how to go about solving the decomposition and composition problems. There are several ways solutions to these two problems might unfold:

- *simultaneously with coding*: the programmer concurrently decomposes, implements, and merges the plans needed for a solution.
- *separate decomposition*: the programmer decomposes the problem completely prior to an implementation.
- *incremental planning:* the programmer alternates between decomposition and composition by identifying a goal, implementing it, then returning for another goal, and so on (also see section 4.1.2.4).

Again, the distinction between these is not as clean as it may appear. Some programmers will, for example, do a separate decomposition at a very high level of abstraction, and then plan simultaneously while coding to satisfy those goals from the decomposition (Pennington and Grabowski 1990). In addition, decompositions may or may not involve the physical production of a design document. This very much depends on the problem and its context. There is evidence that expert programmers spend a large portion of time designing and planning their solutions, many of whom use pseudocode to play this role (Petre 1990). For projects assigned in introductory programming courses, it is generally accepted that design involves the creation of some form of intermediate representation which bridges a problem statement to an implemented program (Pea and Kurland 1983). Effective programmers know the limits of their ability and will either plan "on-the-fly" or choose to write out a solution ahead of time that represents the program decomposition. Many novices, as the next section shows, are not able to make this determination.

2.2 NOVICE PROGRAMMERS

As the previous section laid out, the skills required to be an effective programmer are complex, and the knowledge that underlies these skills is not very apparent. Learning how to program is significant challenge: not only must a new programming language be mastered, but also these less tangible skills. A firm grasp of how particular programming constructs work does not necessarily make the overall task of programming any easier. For example, novice programmers often underestimate the difficulty of writing a program from scratch because they find example programs easy to understand.

2.2.1 What makes programming hard?

It might actually be easier to answer the question "What about programming is *not* hard?" Indeed, novices can struggle with just about any aspect of programming. DuBoulay (1989) provides a top-level categorization of potential trouble spots including: (1) *orientation*, what programming is for and what it means to program; (2) the *notional machine*, the computational model as defined by the programming language; (3) *notation*, defines the syntax and semantics of the language; (4) *structures*, the underlying knowledge structures (i.e., schemata, plans, etc.); (5) *pragmatics*, the skills that drive the building of programs, including decomposing, planning, debugging, etc. Although it is certainly important to address difficulties associated with each of these areas, the focus in this dissertation is on the latter two: knowledge structures and pragmatics.

One of the most frequently reported weaknesses of novices lies in their lack of planning and design skills (Soloway et al. 1982; Pea and Kurland 1983; Pintrich, Berger, and Stemmer 1987; du Boulay 1989; Perkins et al. 1989; Deek 1998; Robins, Rountree, and Rountree 2003). In terms of schema-based theory of programming knowledge, novice deficiencies in planning are easily explained: they struggle to plan because they have not yet "built up" a library of schemata from which to draw – they are essentially breaking ground on this kind of knowledge. Hence, when solving novel problems, novices are not able to retrieve schemata because they do not yet exist (Rist 1995, p. 537), and so they are forced into *creating* them on the fly. One of the claims in this dissertation is that novices need *a great deal of support* during this phase of schema creation – much more than is typically provided.

Possessing a library of schemata is only part of the reason experienced programmers succeed, however. They also have the skills to apply that knowledge. For example, such knowledge is needed to efficiently solve the decomposition and composition problems, both of which are also difficult for novices. Novices have difficulty managing goal and subgoal structure of programs (Pennington 1987). In addition, novices will often implement flawed plans with missing, incorrect, or spurious components, for example (Perkins and Martin 1986). Because the branching factor for non-trivial design tasks can grow large (Soloway, Spohrer, and Littman 1988), the sheer number of decisions that need to be made can make programming uncomfortable for the novice.

Novices struggle with the composition problem as well, even when their decompositions are correct. For example, two plans that handle separate goals may interact in unforeseen ways when integrated into the same program. There are at least three ways such interactions might arise:

- Elements of one plan are duplicated (spuriously) and inserted into another. For example, a variable serving one role could be used incorrectly in a calculation step of another plan.
- Elements of two separate plans are unnecessarily merged. For example, a plan to read in a positive number could be integrated into a primary program loop rather than being implemented as a separate loop.
- Aspects of two plans that must be merged are done so improperly. For example, a plan step that should go inside an existing program loop could be placed after the loop.

In studies of introductory programming students, Spohrer et. al. (1985) found that roughly 65% of the bugs present in their first syntactically correct compilation attempts of a mid-semester project were due to errors of these types. For students at this point in their learning, the programs generally contained the correct plan *parts*, but the real problem was that students were not able to integrate them into a whole solution, at least on their first tries.[1]

[1] The title of the paper (Spohrer and Soloway 1985) summed it up nicely: "Putting it all together is hard for novice programmers."

2.2.2 Novice programmer behaviors

One of the main motivations behind this dissertation is that novice programmers do not generally plan their programs before they attempt to write them, despite the advice (and pleas) of their instructors to do so. Studies have shown that the first step for most novices when writing a program is to type in code (Pintrich, Berger, and Stemmer 1987; Wender, Weber, and Waloszek 1987). The natural tendency is to "get behind the wheel" of a compiler and enter code, often with disastrous results. This is especially true for novices with little or no programming experience. Failure to plan ahead is not surprising given that the most salient weaknesses of beginning programmers revolve around a lack of problem solving and design skills. When faced with a novel problem to solve, then, novices find themselves in a Catch-22: *a variety of problems arise because they did not plan, but they lack the skill and experience to have effectively done so in the first place.*

So, what happens to a novice when they try to write a program without having planned ahead? du Boulay (1989) points out that the difficulties they face "are compounded by the student's attempt to deal with all these different kinds of difficulty at once" (p. 284). Perkins (1989) explains that novices "try to deal with decomposition issues in the middle of coding, instead of planning deliberately in advance" (p. 257). In other words, problem solving and planning questions are addressed simultaneously with the learning of a new programming language and environment. Being forced to deal with so many new issues at one time (i.e., when using the compiler) is not a situation conducive to learning: the central tenet from cognitive load theory (Sweller 1994) is that when too many new topics are being addressed at once, learning is hindered. This issue surfaces again in section 4.1.2.1.

Another particularly detrimental behavior displayed by many novices is the use of program output behavior as the primary means of judging program quality (Joni and Soloway 1986). This view is substantiated by the observation that some novices resort to "tinkering" as a debugging strategy (Perkins et al. 1989). That is, such students repeatedly compile and run programs with only minor changes, hoping each change produces the desired program output. Such behavior is likely a result of poor or nonexistent pre-planning of the program, as well as fragile knowledge of the programming language.

So why don't novices plan? A viable explanation is that novices do not plan simply because they do not know how. Another possible explanation is cultural: novices may view programming and keying in programs as one and the same, and so perceive planning as extraneous to the ultimate goal of producing a working program. It could also be that students are thrown by the deceptive simplicity of the tasks they are asked to program. Because the task is easy to do by hand, they seem to reason, so must be writing a program to perform that task. Yet another possibility is that novices fully intend to plan their programs, but lack knowledge and ability to express the plan in any representation other than code. In general, it seems that novices enter into a programming task without a global picture of their program. Addressing this deficiency appears to be central to helping novices deal with the difficulties of programming.

2.2.3 Why novices do what they do

One of the problems is that introductory programming is often taught from a syntactic perspective in which students learn about programming constructs from example programs and isolated code segments. The focus is on the language constructs rather than the *process* that led up to these final programs. For a novice, it is next to impossible to infer the implicit knowledge that drove the process from a completed example alone. In addition to this, novices also lack an incomplete model of how programs operate and hence, are unable to infer anything about the underlying design decisions of a program. Novices will also create their own explanations for program behavior (Taylor 1999), which further complicates matters since incomplete and incorrect models are learned in these circumstances. Educators and researchers in computer science education have been aware of this problem for decades (Soloway et al. 1982; Soloway 1989; Perkins, Schwartz, and Simmons 1988; Robins, Rountree, and Rountree 2003); but even so, introductory programming courses continue to struggle to produce competency in students (McCracken et al. 2001).

Novices also are hindered by the fact that they tend to lack deep understandings of programming knowledge. In a review of several studies of novices, Winslow (1996) concludes that novices superficially organize their knowledge about programming, have underdeveloped mental models, fail to access and apply the knowledge they do have, and view programming "line by line" instead of as schemata or purpose of code. These are confirmed by numerous studies showing that novices use a bottom-up approach to programming (Rist 1989; Soloway 1989; Wender, Weber, and Waloszek 1987), thinking of programming in terms of small, syntactic units. This is not peculiar to programming: novices in other domains (like physics) display similar behaviors (Chi, Feltovich, and Glaser 1981).

2.3 SYSTEMS FOR NOVICE PROGRAMMERS

A large number of systems have been built in attempts to help novices overcome their difficulties (thorough reviews can be found in Brusilovsky 1995; Deek 1998; Guzdial 2004; Ramadhan and Boulay 1993). A majority of these systems do not *distinctly* address the planning phase of programming, tending instead to support implementation-time activities. Systems in this category effectively encourage "on the fly" planning of programs, which is appropriate for teaching particular language features or for small-scale problems.

However, if the aim is to support novices' problem solving and planning shortcomings, it is generally acknowledged that these concepts should be addressed separately in some form. In this section, a number of systems falling into both categories are reviewed.

2.3.1 Systems supporting on-the-fly planning

2.3.1.1 LISP Tutor As one of the earliest and most famous of all intelligent tutoring systems, the LISP Tutor demonstrated the feasibility of applying the techniques of artificial intelligence and cognitive modeling in a teaching system (Anderson and Reiser 1985; Anderson et al. 1995; Corbett and Anderson 1992). It is an example of a *model tracing* system and was developed as a test of Anderson's ACT-R theory of cognition. The system helps students write LISP functions by providing skeletons that must be filled in by the student. As with all of the ACT-R tutors, the LISP Tutor uses a production rule representation to encode domain knowledge. Students are required to stay on a known solution path at all times and receive immediate corrective feedback when an incorrect action is taken. This usually takes the form of a hint attached to a production involved with the error. Evaluations showed that students using the LISP tutor reached the equivalent or higher levels of mastery in about one-third of the time it takes in traditional programming environments.

A later version of the system reifies problem decompositions by explicitly showing program goals on the screen and scaffolding their use (Corbett and Anderson 1995). In addition, the system relies on examples to help students manage and connect the problem, goals, and ultimately, the solution. An evaluation revealed that LISP programming knowledge was more easily acquired with goal reification and that that the knowledge was more easily generalized when compared with the original system not performing such reifications.

2.3.1.2 Grace Having been developed in industry, Grace (McKendree, Radlinski, and Atwood 1992) stands as one of the few thoroughly field-tested systems for programming available. It teaches COBOL to beginners and experienced programmers alike. Students write programs through the use of menu selections and template completion. Goal and subgoal structure is shown, and help is available at any time via either demand or immediate feedback on errors. An evaluation showed that students tutored by Grace outperformed students who did not use the system on standardized posttests. In addition, they were able to complete more exercises and make progress with less instructor help. A modified version of Grace was later created in response to novice-expert differences observed in the first version of the system (Radlinski and McKendree 1992).

2.3.1.3 GIL One of the main weaknesses of the LISP Tutor was the absence of alternative representations other than the code itself (Corbett and Anderson added some supporting representations in later versions). GIL (Reiser et al. 1992), or “Graphical Instruction in LISP,” provides a graphical view of the process of writing small LISP programs. The system presents “before and after” concrete examples and then helps the student identify the operations that can be used to connect the two. The identified operations are used to write LISP code that performs the same task, but in general. In a similar outcome seen with the LISP Tutor, students using GIL were able to reach similar levels of competency in about half the time.

Far fewer errors were made by GIL students as opposed to those using a traditional, non-intelligent environment. Also, students showed a strong preference for forward reasoning, as opposed to backward.

2.3.1.4 ELM-ART Examples are also the centerpiece in ELM-ART, a web-based, adaptive, interactive textbook for programming in LISP (Weber and Brusilovsky 2001). Using an elaborate overlay and episodic student model, the system provides navigation support, topic sequencing, individualized diagnosis, and example-based problem solving support. Solutions are analyzed via cognitive diagnosis of the code with feedback being generated from this diagnosis. In addition, relevant examples (correct and buggy) are retrieved *en masse* as analogies to support learning. An evaluation of ELM-ART with a previous version of the system (ELM-PE) determined that on a programming task, students using ELM-ART were more successful and took less time to complete all of the lessons (although this may have been due to differences in availability of the two systems).

2.3.1.5 PROUST In an effort to help students debug their programs, Johnson developed PROUST to provide algorithm-related feedback on students' programs (Johnson 1990). Using a large library of programming plans, PROUST builds a *goal decomposition* and generates a tree-like representation of a solution that represents multiple solutions to the problem. Then, the student's program is compared with the generic solution to determine correctness. When the two programs differ, *plan-difference* operators are applied to help the system characterize what is wrong and provide feedback. In an evaluation of PROUST's bug identification capabilities, it was able to localize errors to a high degree of accuracy (94%), however it was not able to identify the precise error in many cases. Also, a complete analysis was required to reach this level of accuracy. In programs that PROUST was not able to completely analyze, performance was much lower.

The intervention provided by PROUST may also be too late for many novices. Early misconceptions have the chance to propagate throughout the whole programming, which may be one reason complete analyses were difficult to obtain in many cases. That is, it is possible that in some cases, students were so confused and so unprepared to write the program that their solution was out of the scope of PROUST. This issue is returned to in chapter 4 when Coached Program Planning is presented, a planning-oriented, or *pre-practice* approach to tutoring.

2.3.2 Systems supporting distinct planning

2.3.2.1 Bridge One of the earliest systems to explicitly support a distinct planning phase was Bridge (Bonar and Cunningham 1988). While using this system, the student enters via menus a high level solution to a problem expressed in a constrained natural language form, via menus. With help available at every stage from Bridge, this solution is iteratively rewritten into more precise forms, ultimately resulting in a working Pascal program. Although student reaction was overwhelmingly positive, no conclusive pedagogical benefits regarding the use of Bridge were ever published. It is possible that the use of menus, rather than open-ended natural language, did not encourage the sort of deep understanding necessary for improved learning. It is also possible that the intermediate language (a puzzle-like representation of plans) was viewed as excessive by students and increased their cognitive load.

2.3.2.2 MEMO-II MEMO-II (Paoloa 2001) also clearly distinguishes planning and design by providing tools for the construction of solution specifications that are independent of any particular programming language. These specifications are used to automatically generate code, thus allowing the student to stay focused on the problem solving aspects of the problem. Deek (Deek 1998) notes that MEMO-II may not be appropriate for novices due to the complexities of the solution specification language. It appears that no evaluation of the system was ever published.

2.3.2.3 GPCeditor The GPCeditor (goal-plan-code, Guzdial et al. 1998) is an integrated CAD workbench that provides tools that help students solve the decomposition and composition problems. With these tools, students first identify goals and plans needed for a solution, then apply plan composition operators, such as *abut* and *nest*, to assemble the identified plans into a complete solution. An evaluation of GPCeditor revealed that it enabled weaker students to produce quality, working programs, but that higher ability students were unhappy with the added restrictions. Most importantly, however, positive transfer was observed to a traditional programming environment. Students demonstrated reuse of known plans and of application of the techniques learned to combine them.

The contrast between the approaches taken in GPCeditor and ProPl is striking. Both systems attempt to teach similar skills – that is, to scaffold students solving the decomposition and composition problems. In addition, both systems work from a plan-based theory of programming knowledge. The key difference lies in the approach taken to elicit and convey this knowledge to the student. In GPCeditor, plans are explicitly named and manipulated with the help of the interface. This follows the original suggestion of Soloway (Soloway and Iyenger 1986). In ProPl, the assumption is that "less is more" and an effort is made to introduce as few new representations as possible. The aim is to teach these ideas *tacitly* through natural language and allow them to develop naturally. The one advantage ProPl may have is that it can be easily "plugged" into any introductory programming course using a procedural paradigm – there is no need to change the compiler or programming language.

2.3.2.4 SolveIt Deek (1999) developed SolveIt as a complete problem solving and programming framework for beginners. The stages of programming identified in section 2.1.2 are all directly and distinctly supported. The system does not provide any intelligent analysis of students' programs, and so is susceptible to the traditional weaknesses of discovery environments (e.g., garden paths). However, evaluations of SolveIt revealed differences both in terms of students' problem solving ability and attitude when compared to students who used the system and a baseline control group.

2.3.2.5 DISCOVER Discover (Ramadhan, Deek, and Shihab 2001) includes a cognitive model of programming and performs model-tracing to track students. Students construct pseudocode solutions under the eye of the tutor, and are required to say on the solution path prescribed by the model. Pseudocode is constructed via buttons which, when pressed, produce pseudocode templates that the student fills in. Discover therefore draws largely on recognition memory, which may hinder student learning.

2.4 CHAPTER SUMMARY

In this chapter, the tasks of programming were presented and literature that addresses the weaknesses and problems of novice programmers reviewed. The main points of the chapter were:

- A *schema* can be thought of as a stereotypical “chunk” of programming knowledge that summarizes a way to achieve a programming goal.
- The cognitive subtasks involved in programming are understanding, planning, implementation, and debugging.
- To write a program, a programmer must identify the goals and schemata needed to solve the problem (the decomposition problem) and assemble the corresponding plans into a proposed solution (the composition problem).
- Novices struggle a great deal with the planning and problem solving aspects of programming.
- One of the reasons for this is that novices have not yet developed a mental library of schemata from which to draw – they must create this knowledge as needed.
- Novices adopt a variety of poor behaviors that make programming even more challenging, such as failure to plan, tinkering with their code, and focusing on output behavior.
- A large number of systems for novice programmers have been designed, but many of them fail to explicitly distinguish between the planning and implementation phases of programming.

3.0 TUTORING AND DIALOGUE

As alluded to in the previous chapter, teaching complex material is often as challenging as learning it, perhaps more-so. This chapter examines a well-known form of teaching: tutoring. A literature review is presented covering the behaviors of expert human tutors, the role of the student during tutoring, and an examination of why tutoring is believed to be so effective. A review of existing educational dialogue systems is also included and the chapter ends with an argument for early tutorial intervention in programming.

3.1 HUMAN TUTORING

It is generally acknowledged that the most effective form of teaching is that of one-on-one human tutoring. Using pre- and posttests, peer tutored students have been shown to score at least 0.4 standard deviations higher than those who receive traditional classroom instruction (Cohen, Kulik, and Kulik 1982). When expert tutors are used, the effect size has been shown to reach 2.0 standard deviations (Bloom 1984). In contrast, the best computer tutors have been shown to achieve effect sizes of about 1.0 standard deviation (Anderson et al. 1995; VanLehn et al. 2002b).

3.1.1 What human tutors do

Tutoring is similar to classroom instruction in many ways: questions are asked, problems are solved, lectures are given, etc. However, there are also some significant differences. Good tutoring is highly interactive – students are more compelled to participate with no other students present (Graesser, Person, and Magliano 1995). Having only one student to monitor, a tutor is more likely to respond to the individual reactions and difficulties of that student. Rather than giving away the answer when the student cannot answer a question, tutors can take actions to help the student make progress (Merrill et al. 1994). When solving problems, tutors are able to help students over impasses and provide targeted feedback (VanLehn et al. 1998; VanLehn et al. 2003). This feedback can take the form of *immediate feedback*, when the student makes an error, or as *demand feedback*, when the student requests help (Merrill et al. 1992). Exactly why tutoring is so effective remains an open question, however (Chi et al. 2001).

3.1.2 Tutor-centered view of tutoring

The tutor-centered hypothesis for why tutoring is effective states that tutoring improves learning because tutors are able to adapt their instruction to the student. This adaptation has been reported to occur in at least four ways:

1. Choosing successive problems according to the student's ability (Bloom 1984).
2. Changing tutoring policies and activities according to affective and motivational states (Lepper et al. 1993).
3. Providing immediate feedback during problem solving to elicit correct steps (Merrill et al. 1992; Graesser, Person, and Magliano 1995).
4. Personalizing explanations to account for the ability and knowledge possessed by the student (Sleeman et al. 1989).

A number of studies have also focused on the tactics used by human tutors, assuming that learning was a function of skilled execution of tutoring tactics. McArthur (1990) analyzed human tutors in algebra and identified a variety of tactics, including some that helped students with task management, understanding goal structure, rule application conditions, and many more. Collins and Stevens (1982) analyzed the strategies of inquiry teachers and identified a variety of strategies including the referencing of examples, identification of domain values or concepts, eliciting of evaluations, and entrapment techniques (i.e., "Socratic" tutoring).

3.1.3 Student-centered view of tutoring

Researchers have also investigated the learning behaviors of students. When students study alone, those who self-explain material (e.g., justify problem solving steps to themselves, make inferences, etc.) learn more effectively than those who read more passively (Chi et al. 1989). This is referred to as the *self-explanation effect*. It is possible to achieve the same effect via *scaffolded* self-explanation, that is, as fostered by the encouragement of a tutor (Chi 1996; Renkl et al. 1998). Furthermore, when pre-test scores are factored out, scaffolded student responses to tutor questions correlate with learning, but self-initiated responses (like questions and self-explanations) do not (Chi et al. 2001, p. 496). This means that the tutor's role in encouraging greater student contributions is responsible for learning beyond the usual benefits seen with self-explanation.

Students who benefit most from tutoring also tend to participate in more significant ways. For example, deeper student contributions during scaffolding episodes correlate with improved learning (Chi et al. 2001, p. 515). Also, more words per student utterance (Rose et al. 2003b) and the ratio of student words to the tutor's (Core, Moore, and Zinn 2003) both correlate with learning gains. The general implication of these findings is that tutors should seek to maximize student participation and elicit meaningful contributions from students.

3.1.4 Pre-practice intervention in tutoring

The classic view of the tutor is that of sitting next to the student while s/he works on problems. While the student works, the tutor observes, answers questions, and intervenes when it is deemed necessary. Tutoring can take place at different times, however. Another option is to tutor *reflectively* by reviewing a problem after it has been solved. Activities in reflective tutoring include things like highlighting good steps, identifying impasses, correcting conceptual misconceptions, and discussion about how the problem solving could have been improved (Katz, Allbritton, and Connelly 2003). Finally, tutoring can also occur *before* problem solving to help "set the stage" for student before they actually begin working on the problem. This is depicted in figure 3.1. Although researchers have established the pedagogical benefits of reflective tutoring tutoring in physics (Katz, Allbritton, and Connelly

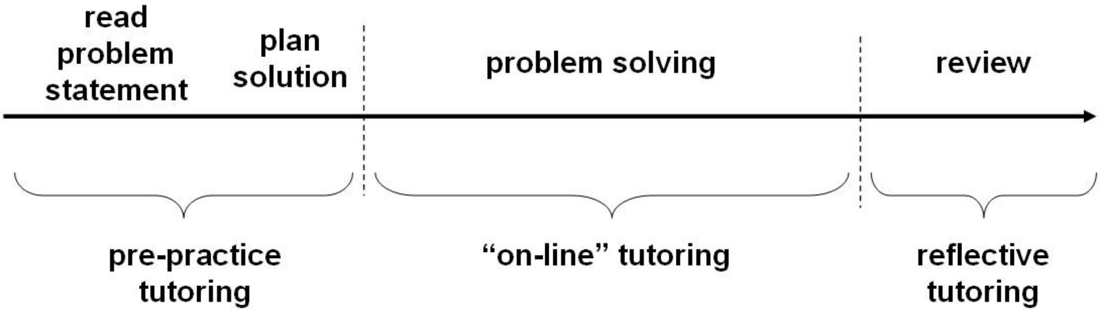


Figure 3.1: Three categories of tutoring.

2003) as well as with collaborative science learning (White, Shimoda, and Frederiksen 1999), much less work has been done with early, pre-practice tutorial intervention. Part of the aim of this dissertation is to explore the value of this type of tutoring.

A related notion, *preventive tutoring*, has received substantial attention in the reading community. The aim of preventive tutoring is not to prepare the student for independent reading sessions (where reading plays the role problem solving), but rather to intervene *in the early stages of learning*. The idea is to increase the pedagogical presence when reading skills are first being developed, thereby reducing the chance that errors will slip by and become more ingrained. In the 16 studies reviewed by Wasik (1993), nearly all of them report success, some with substantially large effect sizes. In sum, for the acquisition of reading skills, preventive intervention works exceedingly well.

As chapter 4 will suggest, preventive tutoring does indirectly apply to the pedagogical approach presented in this dissertation. As in the reading studies, this research targets novice programmers at the time they are acquiring programming skills. The general aim is to reduce the amount of time they are left alone to pursue the poor habits discussed in the previous chapter. However, because the goals here are more focused – to provide help on individual assignments – the term *pre-practice* tutoring is used in this dissertation. It is a fit with programming since novices do not tend to plan on their own anyway.

3.2 REVIEW OF DIALOGUE-BASED EDUCATIONAL SYSTEMS

The best intelligent tutoring systems improve student performance by 1 standard deviation. For many of these systems, however, shallow learning is a big concern. Some students will learn how to use the system successfully rather than learning the domain. It has been suggested that natural language dialogue is a possible solution to this problem. As discussed earlier, by asking questions and eliciting answers, it is believed by many that deeper learning will result. This belief has motivated the development of a number of dialogue-based tutoring systems. In this section, some historical and modern systems falling into this category are reviewed. A review of the few dialogue-based systems for programming that exist is also presented.

3.2.1 Early systems

In the 70's and early 80's, researchers began to recognize the potential of natural language dialogue in educational systems. Examples of such systems included Carbonell's geography tutor SCHOLAR (1970), Collins and Stevens' Socratic tutor for rainfall processes WHY (Stevens and Collins 1977; Collins and Stevens 1982), and Burton and Brown's series of simulation-based systems SOPHIE I, II, and III (Brown, Burton, and Kleer 1982). Extensive reviews of these systems can be found in (Wenger 1987). All were seminal systems and laid much of the groundwork for modern dialogue-based educational systems; however, because of the limited success of natural language processing at the time, the dialogue capabilities of these systems were limited. The resulting dialogues tended to be unrealistic and students were commonly restricted to short answers.

3.2.2 Modern systems

A rebirth of dialogue-based educational systems occurred in the late 90's. Although the rigidity of early systems was still present to some extent, corresponding technological improvements in natural language processing and statistical approaches to AI (Jurafsky and Martin 2000) helped make dialogue systems more robust and usable.

3.2.2.1 Why2-Atlas Why2-Atlas (VanLehn et al. 2002a) is a system that poses qualitative, conceptual physics problems and analyzes student essays that are given as answers. Remediation dialogues are used when problems are detected in the essay, like missing concepts or evidence of misconceptions. The dialogues generally try to help the student recognize these errors through entrapment (Socratic) means, or by the use of example situations. Students have a chance to revise their essay and resubmit until the essay adequately answers the original question. Why2-Atlas attempts a deep linguistic analysis to produce a first order representation of the content of the essay (Rose et al. 2001), then uses this representation along with an abductive theorem prover (Jordan and VanLehn 2002) to determine the issues that should be addressed with dialogue. Evaluations of the system have shown mixed to moderate posttest gains for students using KCDs versus those who read mini-lessons (Rose et al. 2001; Rose et al. 2003a).

3.2.2.2 Circsim-Tutor To help first-year medical students learn about the reflex control of blood pressure, Evens and her colleagues have developed Circsim-Tutor (Evens et al. 2001; Glass 2001). Students using the system fill cells in a table that represent their predictions of changes in parameters, such as heart rate and blood pressure. When errors are detected in the predictions, Circsim-Tutor conducts a tutorial dialogue aimed at helping the students understand the mistake and revise their prediction. Many different types of errors are recognized, and for each, a pre-authored line of reasoning is executed as a dialogue plan. Although only closed-answer questions are ever asked and student answers are generally short, this is appropriate because of kinds of problems it targets in the domain of cardiovascular physiology. For example, a typical question asks the student to predict if one variable should increase or decrease when given information on other, predictive variables in the model.

3.2.2.3 AutoTutor AutoTutor was developed to simulate dialogue patterns of unskilled and skilled tutors (Graesser et al. 2001). Students are asked *how*, *why*, and *what-if* questions to elicit domain facts and concepts from a student. The system employs Latent Semantic Analysis (LSA) to determine which concepts are present in a student's utterance

(Graesser et al. 2000), and uses a variety of techniques to encourage the student to contribute to the dialogue. For example, if a student's answer matches on a subset of the expected concepts, AutoTutor may return with a pump, such as "What else?" Other tactics include hints, completion questions, and gesturing (an avatar is used to deliver the tutor's contributions). All questions asked are closed-answer in that specific answers are expected for the tutor's questions. Expected answers are spelled out in a *curriculum script*, which includes answer *aspects* that the tutor seeks to elicit. Some buggy answers and responses are also included. Variance in student answers is automatically handled by LSA and the provision of a variety of answers on the curriculum script. Versions of AutoTutor exist for computer literacy, conceptual physics, and research methods.

3.2.2.4 Geometry Explanation Tutor The Geometry Explanation Tutor is an extension to the Geometry Cognitive Tutor that, in addition to adding lines to a proof, requires students to give English explanations of those steps (Aleven, Koedinger, and Popescu 2003; Aleven et al. 2003). To understand student explanations, a robust parser is used to produce semantic representations that are in turn, classified by a LOOM knowledge-base. Based on the classification, feedback is generated by the system to elicit the precision necessary to correctly explain geometry theorems. For example, a student might suggest the base angles of a triangle are equal, but forget to say that this is true only for isosceles triangles. Since at each step the student must explain the theorem used, all questions posed by the system are closed-answer in that a precise list of theorem conditions must be stated by the student. In a controlled experiment comparing the explanation version of the system against the original, and with time fixed, the explanation tutor led to improved performance on the posttest.

3.2.2.5 BEETLE In an effort to go beyond the simple dialogue management techniques used in many dialogue-based tutoring systems, such as finite-state-based control and form-filling, BEETLE adopts a 3-tier planning architecture inspired by those used in robotics (Zinn, Moore, and Core 2002). It consists of an *interpretation* module to do syntactic, semantic, and intentional analysis of student utterances, an *update* module that maintains dialogue context, and finally a *response generation* whose job is to determine appropriate tutorial moves. Three levels of planning occur: deliberative (high-level of abstraction), context-driven (sub-dialogue planning), and action execution (performing primitive acts). This model permits tutorial reasoning at different levels of abstraction and allows the tutor to use more advanced tactics. BEETLE is the prototype system built to validate and experiment with the architecture (Zinn et al. 2003). The domain is electronic trouble-shooting and the goal of the tutor is to walk the student through problems, eliciting next steps, elaborating the reasons for taking certain steps, and correcting misconceptions. At the time of this writing, the system was not yet complete and no evaluation had been conducted.

3.2.3 Dialogue-based systems for programming

Only a few dialogue-based systems have been developed for the domain of programming, and none of these seems to target the planning or problem solving aspects of programming. This may be due to the fact that generalized domain knowledge for programming is notoriously complex. For example, the PROUST system (not a dialogue-based system, see section 2.3.1.5) dealt directly with algorithm-related errors. After a massive effort to analyze student programs and build the system, it was only able to tutor students on two programming problems.

3.2.3.1 Program Enhancement Adviser To demonstrate theories of interactive explanation, Moore (1995) developed the Program Enhancement Adviser (PEA) as a "test bed in which to explore issues of interactive explanation generation" (p. 13). It helps programmers improve the style of their LISP code by suggesting transformations that help with readability and/or maintainability. Using dialogue, PEA gives advice to the student on changes to make, such as the benefits of using `SETF` instead of `SETQ`. Students can ask for elaboration on why the changes are being suggested, what certain terms mean, and for follow-up explanations. Since the goals of the research were to test a theory of interactive explanation and build a prototype, only a non-trivial subset of recommendations were implemented in the system. It was not designed for broad coverage of LISP, and thus underwent no pedagogical evaluation.

3.2.3.2 Duke Programming Tutor The Duke Programming Tutor (Keim, Fulkerson, and Biermann 1997; Keim 1997) provided spoken dialogue assistance to novice programmers in the debugging of simple Pascal programs. Students could initiate a dialogue when they had errors to ask the system for advice. For example, if asked "What is wrong with my program?", the system might respond "There is something wrong with this writeln statement." It appears that the goal of this system was to present the student with compiler error feedback in a more palatable form along with advice about how to fix the errors. In addition, it also was used to develop and test new algorithms for the taking and releasing of initiative. It appears no evaluation of the system took place, or that it ever made it beyond the prototype phase.

3.2.3.3 GENIUS Deemed as an exercise in "ignorance-based reasoning," the GENIUS system attempted to use ELIZA-like interactions to help students repair syntax errors in their programs (McCalla and Murtagh 1991). The motivation behind this system was to keep the student talking and hope they would be able to resolve their own errors. Understanding is limited to yes/no questions and "I don't know" responses. The compiler output is analyzed to determine what content to present, which is given to the student after a number of pattern-matched exchanges. GENIUS was not able to provide help for 60% of the requests in one study (also see Deek 1998).

3.3 CHAPTER SUMMARY

In this chapter, literature on tutoring was reviewed as well as that on several dialogue-based tutoring systems, old and new. The main points of the chapter were:

- One-to-one human tutoring is generally acknowledged as the best form of teaching.
- Although it is not known precisely why tutoring works so well, some believe it is due to the adaptive nature of tutoring.
- Studies have shown that tutors adapt by selecting appropriate problems, detecting the affective state of the student, providing immediate feedback to keep the student on a productive path, and personalizing explanations for the student.

- Tutoring can take place in three ways with respect to problem solving: before (prepractice), during ("on-line"), or after (reflective).
- The self-explanation effect occurs when students reason about the material they study, make connections, infer properties, etc. Students who self-explain learn more effectively, and tutoring has been shown to encourage self-explanation in students.
- Students who contribute more during dialogue also learn more, and so the goal of dialogue-based systems is often to elicit as much as possible from the student.
- A number of dialogue systems for educational purposes have been developed, the first of which surfaced in the late 70's with a second wave arriving in the late 90's. This "rebirth" coincided with advances in natural language processing and dialogue systems as well with research showing that tutoring could be effective at eliciting self-explanations, of which natural language is necessary to perform on a computer.

4.0 TEACHING THE TACIT KNOWLEDGE OF PROGRAMMING

The problem addressed by this dissertation is *how to scaffold the development of the tacit knowledge of programming through the use of natural language tutoring.* In this chapter, *Coached Program Planning* (CPP) is presented as a proposed solution to this problem. CPP began as an exploratory effort to find out how novice programmers would talk about solving programming problems in the abstract and *before* they had a chance to attempt to solve them. Based on the intense difficulties they face with programming, it made sense to provide help that started with first exposure to the problem. After I conducted a series of computer-mediated tutoring sessions, CPP started to take shape. The purpose of this chapter is both tell this story and layout the details of the tutoring model. All of the data presented here played a major role in the design of ProPl, which is the topic of chapter 6.

In many ways, this chapter is the centerpiece of the dissertation because it addresses most directly how tutoring can be used to open up the concepts of programming to novices. Dialogue examples are presented to demonstrate fundamental aspects of CPP. This chapter can therefore be viewed as a "how-to" guide for readers interested in tutoring novice programmers.

4.1 PRELIMINARIES

Before presenting the details of the CPP tutoring model, this section provides some history and pedagogical background for the rest of the chapter.

4.1.1 CPP history: a personal account

The roots of this research effort surfaced in late October of 2000. I had been teaching introductory programming courses for nearly 5 years at that time, and had formed the belief that novices needed a different kind of help to maximize the value of the time they spent programming. In a document describing my initial thoughts on what kind of system to build, I wrote the following:

> The basic idea behind this proposal is to build an ITS that tutors on two issues: problem comprehension and decomposition. Success on both measures would mean a student should know (beyond just an observational understanding) what exactly a problem is asking, and secondly, should have an initial problem decomposition establishing a sequence of attainable programming goals ultimately solving the problem.

I had defined *observational understanding* as simply the ability to read an example of some task and follow the steps. It was the first stage in an early model of programming ability I developed. The higher levels included the ability to perform the task (say, on paper), and ultimately the ability to write an algorithm to perform the task. At the time, I had not committed to natural language (or, for that matter, any of the decisions laid out in the next section). The early aim was to simply build a tool that helped novices understand *what* they were trying to do, and have an initial idea of *how* to proceed. This aspect was motivated largely by informal reports of students not knowing what to do once sitting in front of a computer.

About a month later, I began creating scenarios for several problems (most namely, Hailstone and RPS). It was at this time I realized that working directly in some specific programming language would only increase the chances of syntax and compiler-specific issues would only muddy an already complex domain (i.e., what happens anywah when novices enter into an implementation prematurely. This view is substantiated by research in cognitive load (Sweller 1994), and so the best alternative was to use pseudocode, especially given its long history as a planning tool and the inherent absence of syntactic rules. The early scenarios I constructed looked surprisingly similar to actual CPP dialogues I would eventually collect, including discussions about what to do next, how to figure out termination conditions on loops, and so on.

A few aspects of these crafted scenarios did not make it into the eventual CPP model. The two most salient of these were:

- *abstract tiles*: allowing the student to roll-up a bunch of steps into one “superstep”, and
- *reflective plan reification*: creation of a written plan (to take home) *after* the pseudocode is complete.

The abstract tile idea was dropped quickly because of fears that students would not like it in such small programs (maybe I was wrong). The reflective plan reification did, however, make it into the pilot study (discussed below).

To gather some real student data, a pilot study was run in the summer of 2001. A simple environment was built (figure 4.4, page 58), IRB approval granted, and then I advertised. 11 students volunteered out of about 45 in the class. Interestingly, the gender split was 8 females and 3 males, but the class had roughly the inverse split (80% male). The students were asked to not look at the problem statement until they came in for their scheduled tutoring session.

Sessions were simple: students came in, were given a brief description of the environment, and then started typing their answers to questions in the chat window. As the discussion proceeded, the pseudocode grew, and at the end, they had a pseudocode plan to take home with them. In this study, reflective plan reification was performed, but *students had trouble understanding why this was necessary given that the pseudocode was already complete.* It was awkward to tell students that there was more work to do even when the pseudocode was complete. The way it worked was to first remember how the program was written, and then highlight certain steps that contributed to each identified goal. Once done, this produced a sequence of gradually larger programs, each consuming another goal. When I began to notice that students were only requesting printouts of the final pseudocode (and not the list of derived programming goals), I realized that the reflective goal reification idea was not going to last. Students viewed it as superfluous.

The most important outcome of the pilot study, however, was the corpus. With this collection of student utterances and this record of my own tactics to help students build the programs (indeed, I had used similar tactics over the years to help students write real code), some more practically useful observations could be made (this is the topic of section 4.3.1). CPP began to take shape in the fall of 2001 and spring of 2002 with more subjects.[1] The rest of this chapter details the model and the issues involved in its design.

4.1.2 Pedagogical underpinnings

Clearly, my early beliefs and research activities had a huge impact on CPP. It is true that development of any pedagogical approach or tutoring system requires navigation through a huge space of choices, each of which could be the topic of a research study. For example, what type of interaction is best? Should intervention take place before, during, or after problem solving? How should domain knowledge be presented to the student? This section presents four principle decisions that underly CPP:

1. tutor the student *before* actual problem solving (with a compiler) takes place
2. use *natural language dialogue* as the primary mode of interaction
3. present program code as *natural-language-style pseudocode*
4. teach a *staged* approach to program development

The primary purpose of this section is to defend these choices as reasonable and provide the context for the remainder of the chapter. The thesis is that natural language tutoring is effective at making the tacit knowledge of programming more accessible by novices than it normally is. Therefore, in the evaluation presented in chapter 7, the only independent variable is the use of dialogue: a dialogue group is compared with a read-only group – the other three choices remain fixed.

4.1.2.1 Argument for pre-practice intervention As discussed in section 2.3, most systems developed for beginning programmers are intended for use *while the student is engaged in the act of programming.* A small fraction of these systems provide assistance for the planning of programs, and none provide help with *only* planning in mind. In general, most systems are geared towards providing support during practice (e.g., Anderson et al. 1995;

[1] This was also about the time it got its name.

Gertner and VanLehn 2000). As a result, less is known about the impact pre-practice tutorial intervention might have on students when they sent off to solve problems independently. One example of such a system is the Hierarchical Analysis Tool (HAT) which provides the student with a set of problem-independent menus for use before solving physics problems (Dufresne et al. 1992). The goal is to help the student identify relevant physics principles to think about applying and set the stage for principle-grounded problem solving. Several experiments with HAT found that it helped novices judge solution similarity and solve problems more effectively. The fact that HAT was found to be beneficial for physics learners is suggestive that CPP could do the same for beginning programmers.

For programming, pre-practice intervention has appeal because of the fact that programming is a design task – it can and should be done in stages. As discussed in section 2.2.2, novices do not generally engage in any meaningful planning activities. One of the many negative outcomes of this behavior is summarized well by Perkins, who also casts a vote for early intervention:

> Finally, students face trouble in breaking problems down because they often try to deal with decomposition issues in the middle of coding, instead of planning deliberately in advance. Instructional intervention which encourages pre-planning might help. (Perkins et al. 1989, p. 275)

Novices force themselves into planning on-the-fly and so must make key problem solving decisions at the same time they are learning about the programming language, compiler, editor, and so on. According to cognitive load theory (Sweller 1994), too many *new* issues in working memory at once hinders learning. This happens because the interactions between the components is too much for a student to manage. Many novice programmers have little chance on their own to avoid this situation.

In studies of cognitive load theory, it has been shown that an effective remedy is to teach new topics in a domain *successively* rather than simultaneously, whenever possible (these are summarized in Sweller 1994). This reduces the interactions between the components, and therefore promotes learning. In programming, since students are not able to do this naturally, the suggestion is that pre-practice tutoring is needed to help students recognize and act on the distinction between planning and implementation. Such support would allow problem solving issues to be addressed separately from programming language and compiler issues, thereby reducing their implementation-time cognitive load. In sum, from a cognitive load perspective, pre-practice intervention seems to be a good fit.

4.1.2.2 Natural language and programming The relationship between natural language and programming has been studied by a number of researchers. A fundamental problem for learners of programming has to do with the incongruities that exist between these two classes of languages:

- There are "serious 'cognitive mismatches' between the solutions provided by naive programmers *intended for other people* and the solutions required for effective computer programming" (Miller 1981, p.194).
- Novices often impose commonsense meanings of words onto their use in a programming language. For example, many students interpret *then* in a chronological sense ("first do this *then* that") rather than a conditional sense (Bonar and Soloway 1989).
- Some students will introduce new vocabulary that is not part of the of the programming language they are using (Bruckman and Edwards 1999).
- There are often expectations and assumptions that would be reasonable in a conversation with another human, but completely out of the scope of a compiler or interpreter (Bruckman and Edwards 1999; Miller 1981).
- When asked to describe algorithms in free-form natural language, the "code" produced differs in fundamental ways from the semantics of common programming languages. For example, beginners often prefer a *daemon-like* style of loop that constantly monitors the termination condition rather than the usual one-time check that accompanies each iteration (Pane, Ratanamahatana, and Myers 2001). This has also been observed as a common novice misconception of how loops work in real programming languages (Bonar and Soloway 1989).

In short, there seems to be a kind of *negative transfer from natural language to programming* and it has the potential to be a major hindrance to learning. Students confuse the semantics of the languages and are often unable to "reset" the context when switching between them.

One of the challenges of teaching programming, then, is to help raise the awareness in students of *what it means to program* and how it differs from natural communication skills.

From the perspective of programming educators, it is generally believed that many popular languages (e.g., C, Java) function quite poorly as a first language. This is indeed one of the great, and perhaps misplaced debates of computer science education. At a conceptual level, the differences between Pascal, C, Java, etc. are minimal. Mismatches with natural language, such as those mentioned above, exist with all popular programming languages. In attempts to make programming less difficult for beginners, a number of researchers have therefore advocated the design of programming languages that better match natural ways of thinking and communicating. These efforts have included more intuitive control structures (Pane 2002), increased use of natural vocabulary (Bruckman and Edwards 1999), and the use of more intuitive syntax (McIver and Conway 1999). The general idea is to "close the gap" between a problem statement and a programmed solution by limiting the cognitive hurdles imposed by traditional programming languages, thereby making it easier for novices to express their solutions in the desired language.

The major drawback of this approach is that teachers and departments are reluctant to use languages in their introductory courses that are not mainstream. Also, changing the language of an introductory programming course is likely the single most dramatic change imaginable. The ramifications of such a change often extend well beyond the introductory course: it is common for subsequent courses in a curriculum to depend on knowledge of a certain language. In practice, then, it does seem not practical to overhaul introductory courses this radically. This chapter lays out an approach that is intended to remain sensitive to the concerns of those who believe commonsense notions of language should be recognized, but also to the natural resistance to replace traditional programming languages with pedagogical ones.

So is there anything about natural language that might *help* novices rather than get in their way? Aside from the obvious fact that negative transfer from natural language occurs anyway, there are in fact several reasons to apply natural language tutoring to the problem of teaching the tacit knowledge of programming:

- Scaffolded self-explanation with the encouragement and support of a tutor, improves learning (Chi 1996, see section 3.1.3).

- Encouraging students to elaborate ideas in their own words is known to facilitate learning of technical material (Mayer 1980).
- It is easier to connect to students' existing knowledge by using natural language because it is familiar to them (Mayer 1989).

Although some aspects of natural language certainly present complications, it *is* the medium humans use for communication and it *does* overlap in many ways with the semantics of modern programming languages. The basic problem that novices face is that they generally lack the experience and ability to understand the distinctions between languages for human communication and those for communication with machines. This is where tutoring can fit in: by establishing a tutorial presence with students when they are first encountering the challenges of programming, it may be possible to help novices develop an understanding of the differences and stave off some of the misconceptions that naturally arise.

Lastly, why dialogue? Why not tutor directly in the target programming language or in some specialized design environment? One reason is that there are pedagogical reasons to elicit student contributions in their own words (see section 3.1.3). Many of the systems discussed in section 2.3.1 present menus or buttons that allow the student to work in the problem space. A common problem with such systems is that students learn to focus more on where to click or what action to take to make progress.[2] This means that *recognition* memory is being tapped instead of *recall* memory, which is necessary for deeper learning. In general, for deeper learning, systems should eschew menu or button-based interfaces in favor of those that require that students provide answers in their entirety (this argument also appears in Rose et al. 2001).

The most obvious way to achieve this is to have the student enter the actual problem solving steps (lines of a program, in the case of programming). This approach has been used many times over, including in the domain of programming (Anderson et al. 1995; McKendree, Radlinski, and Atwood 1992; Reiser et al. 1992). Starting with a specific programming language, however, is contrary to the CPP aim of helping novices plan ahead and develop a problem-driven, more abstract view of programming. So, the use of natural language

[2] Such systems also often make it easier for students who try to "game the system" in order to finish their assignments. That is, some students will click aimlessly just trying to get through the material.

dialogue stands as a compromise between requiring the use of recall memory and avoiding the specifics of a particular programming language.

Another potential solution is to introduce a specialized intermediate language that can be used to express a solution design. This was the approach taken in Bridge (Bonar and Cunningham 1988). The basic problem here is that any new language, no matter how intuitive, will increase cognitive load. Although it possessed intuitive appeal, the puzzle-piece planning language used in Bridge may have been viewed as excessive for some of the test subjects. New (and superfluous) languages unavoidably introduce new learning hurdles. However, natural language is already familiar to the student. Students can try to express the concepts of programming in their own words and, hopefully, make connections to existing knowledge more readily. And so, CPP calls for natural language dialogue to be the primary mode of communication with the tutor, be it a human or computer.

4.1.2.3 Pseudocode Pseudocode is a well-established approach to teaching novice program design because of many of the issues discussed above. Pseudocode is not intended to be executed by a computer, but rather to capture key aspects of an algorithm being created. At one extreme, Shackelford promotes the exclusive use of (formal) pseudocode and no contact with a compiler at all during a semester (Shackelford 1998). Others have argued for a gradual introduction of pseudocode along with a particular programming language (Lee and Phillips 1998). The approach taken in CPP fits in with the latter.[3] The style of pseudocode used in CPP tutoring is informal in an effort to better match the language of the student (it is similar to that adopted in Robertson 2000). An example CPP pseudocode solution appears later in this chapter (section 4.2.2, page 50).

A second important motivation to using pseudocode is that it allows the tutor to avoid talking about programming language specific topics. Beginners are known to think of programming in a very concrete, low-level way (Winslow 1996). For example, common responses from novices when asked about what to do next in a (Java) program are “use println()” or “Math.max().” By using pseudocode as the target of the problem solving, it is easier to work at an abstract level and encourage students to *not* think about a particular programming

[3] Although, it could certainly be applied in a pseudocode-only style of class as well.

language. Also, most of the concepts of programming are independent of any language (e.g., repetition, conditional execution, and so on); by using pseudocode, language-independent learning is indirectly supported (although CPP is directed at the procedural or imperative paradigm, at least at this time).

4.1.2.4 Staged design Rather than provide direct support of top-down design, a *staged* style of program development is supported in CPP. In top-down design, the programmer first identifies all programming goals, then iteratively refines these goals until they bottom out into program steps. Staged design also prescribes the identification of goals, but in a slightly different way. The idea is to identify a simplified sub-problem of the original problem (which is often the same as a goal identified in top-down design), then write a program that solves that sub-problem. At this point, the programmer revisits the problem to determine the next goal to achieve, then returns to the program to solve that goal. This process is repeated until the complete problem is solved. Plan merging (discussed in section 2.1.3) therefore occurs incrementally, with plans being integrated into the whole solution one at a time.

One of the more subtle skills that underlies a staged approach is learning *what to ignore* in the problem statement. In this sense, staged design is similar to *focal design* (Rist 1995) in which the programmer first identifies the "simplest, most basic action" involved in a solution, then builds around that (p. 537).[4] In CPP, however, no commitment is made to whittle a problem down to a "most basic action," but rather the more vague notion of simplifying the original problem is adopted.

There is empirical support for the appeal of staged design to novices. In Spohrer and Soloway's (1985) analysis of the first syntactically correct programs submitted by novices, they were forced to drop 25 programs (out of about 160) because these students chose to write small programs that did not attempt to solve the whole problem. They were interested in identifying the interactions between plans and the problem this posed for their students. Since these 25 students had not attempted to implement all plans in the problem at once,

[4] Two other names for staged design found in the literature are *incremental development* and *programming by improvement*.

there was much less of a chance for interaction errors. In many cases, these partial solutions were in fact the first step of a staged approach. Spohrer and Soloway surmised that this behavior was one way some of their students coped with the complexities presented by the composition problem because they realized it was easier to merge plans incrementally (one at a time) rather than some larger number of plans all at once.

Support of the staged approach in the tutoring model presented here is not, however, staged in the truest sense because of a commitment to pre-planning. That is, students are not "released" after talking about a goal and plan to actually go write the program. Instead, the tutoring session is a kind of dry run of the planning process which uses pseudocode as the target language. In this sense, the tutoring is taking the role of an *advance organizer* that primes a student for an independent implementation phase by situating their pending (new) experience in context that is meaningful and accessible to them (Ausubel 1960). The tutoring session is intended to help the student develop an improved understanding of both the problem and how to go about solving it.

4.2 TARGETED PROBLEM TYPES

CPP works best on a certain class of problems, which is the topic of this section. Also, to illustrate the program planning model embedded in CPP and provide a context for dialogue examples throughout the rest of the dissertation, this section discusses the *Hailstone problem*,[5] a typical assignment given to students four or five weeks into an introductory programming course. Two other problems, *Rock-Paper-Scissors* (RPS) and *Count/Hold* (CH), also played prominent roles in this research. Descriptions of them can be found in appendix A.

[5] I first encountered this problem in Doug Cooper's popular textbook *Oh Pascal!*, which, interestingly, was also one of the earliest textbooks to reify plan-like knowledge in a way suitable for novices.

4.2.1 The Hailstone problem and other knowledge-lean tasks

Introductory programming assignments can usually be classified as *knowledge-lean*; that is, "problems that require very little knowledge to solve them" (Robertson 2001, p.8). Such problems are also very typically easy to perform by hand, although not necessarily easy to solve. Some simple examples include dominoes, tic-tac-toe, and Go. Sometimes the tasks underlying these problems are overt, while in others, although still trivial, involve subtle assumptions and hidden steps that are easy to overlook. Knowledge-lean problems falling into this category often require some piece of insight to solve.

The reason knowledge-lean problems are so attractive for introductory programming instructors is that they allow more time to be spent on the programming and problem solving aspects of solving it – little background work is necessary. A drawback, however, is that knowledge-lean tasks are often *deceptively simple*: novice programmers sometimes believe that "easy to understand" implies "easy to program." Unfortunately for them, there are often insights that arise when attempting to solve a problem algorithmically that *do not* appear when a problem is solved by hand. It therefore requires a deeper level of understanding and ability to construct an algorithm to perform a specific task.

A good example of a knowledge-lean problem that is particularly challenging to program is the classic game of Rock-Paper-Scissors (RPS). Most people already know how to play, and if not, can easily be taught. The act of playing a game is essentially a simulation of the desired program behavior. Performing such a simulation is often referred to as a *hand calculation*, terminology introduced by Spohrer (Spohrer and Soloway 1985). There are a large number of issues that arise when writing a program to play RPS that are not apparent from simply playing the game. For example, how does the computer pick? How do you determine a winner? How do you represent the three choices? An ability to play a game by hand does not obviously help a novice when faced with these questions. In fact, based on the earlier discussion of novice behaviors, it seems unlikely that most novices will even generate these questions until they open up their favorite editor.[6]

[6] Yet another example often cited in the literature the averaging problem (e.g., Shackelford 1993). College students rarely have a problem computing an average of a list of numbers by hand, but most struggle to write a program that can do it.

The January 1984 issue of *Scientific American* contains an article describing an interesting sequence of numbers known as the *Hailstone Series*. The series is formed like this:

1. Pick a positive integer.
2. If it's odd, triple the number and add one.
3. If it's even, divide the number by two.
4. Go back to 2.

Although the numbers usually bob up and down, they eventually they reach a repeating "ground" state: `4 2 1 4 2 1 ...` This has been proven for every number up to about `1.2E12`.

You are to write a program to generate a Hailstone series for initial values entered by the user. Your program should answer the following questions after the ground state has been reached:

- How many items are in the sequence?
- What is the largest number the sequence reaches along the way?

You program should stop counting once any member of the ground state (`4 2 1`) is reached.

Figure 4.1: The Hailstone Problem Statement

RPS is actually too complicated to be given early in an introductory class, however. As such, another knowledge-lean problem that we will consider, and draw from heavily, is the *Hailstone problem*, shown in figure 4.1. It involves an interesting, somewhat unpredictable sequence of numbers. The problem statement also asks the student to count the items in the sequence and find the largest value encountered. For most, it is trivial to produce a sequence on paper since only very basic math skills are required. Counting the items and finding the largest are similarly very easy to do – all students in involved in this research were able to do so easily with the only errors being minor slips. As with RPS, however, writing a program to solve Hailstone is challenging.[7]

[7] About half of the students in the study spent at least four hours working on their solution. When asked what was hard about it, most responded that it was the first assignment that required the use of several different programming constructs together in one program.

4.2.2 An illustrative solution to Hailstone

A programmed solution to the Hailstone problem requires the use of several variables, a conditional statement for the update rules, a loop to produce a sequence, a counter, and a test to track the largest value. To concretize the nature of the knowledge that underlies CPP, Hailstone is solved below using a staged approach and with pseudocode.

The first step for a student is to read the problem statement (figure 4.1). Next, CPP prescribes a hand calculation. Starting with the initial value of 11, for example, would produce the following sequence and answers:

- 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.
- the length is **13** and largest is **52**.

The count should include the initial value and the first value reached in the ground state (which will be 4, unless another value in the ground state is used as the initial value). To perform this hand calculation, the student only needs to follow the problem statement and apply basic arithmetic skills.

As with most programs, there are a number of potential "good" solutions to the Hailstone problem. However, because it is a relatively small program, many of the possible variations between different solutions are inconsequential, such as the relative locations of independent steps within a loop. The solution presented below is simply a form of the most commonly produced solution from my own experiences with students solving this problem. So, to begin solving the Hailstone program using staged design, the first step is identify an appropriate sub-problem that will act as the first program goal. By inspecting the problem statement, it is clear that step 1 says to "Pick a positive integer." Thus, for the program, the first goal is to read in the initial value from the user (initial-value).[8] The schema for this goal is simple, consisting only of only a prompt and read. It should be noted that the problem statement does indicate that this value should be positive, and so the actual program should ultimately check for this. Currently, this is left out of the code to reduce cognitive load.

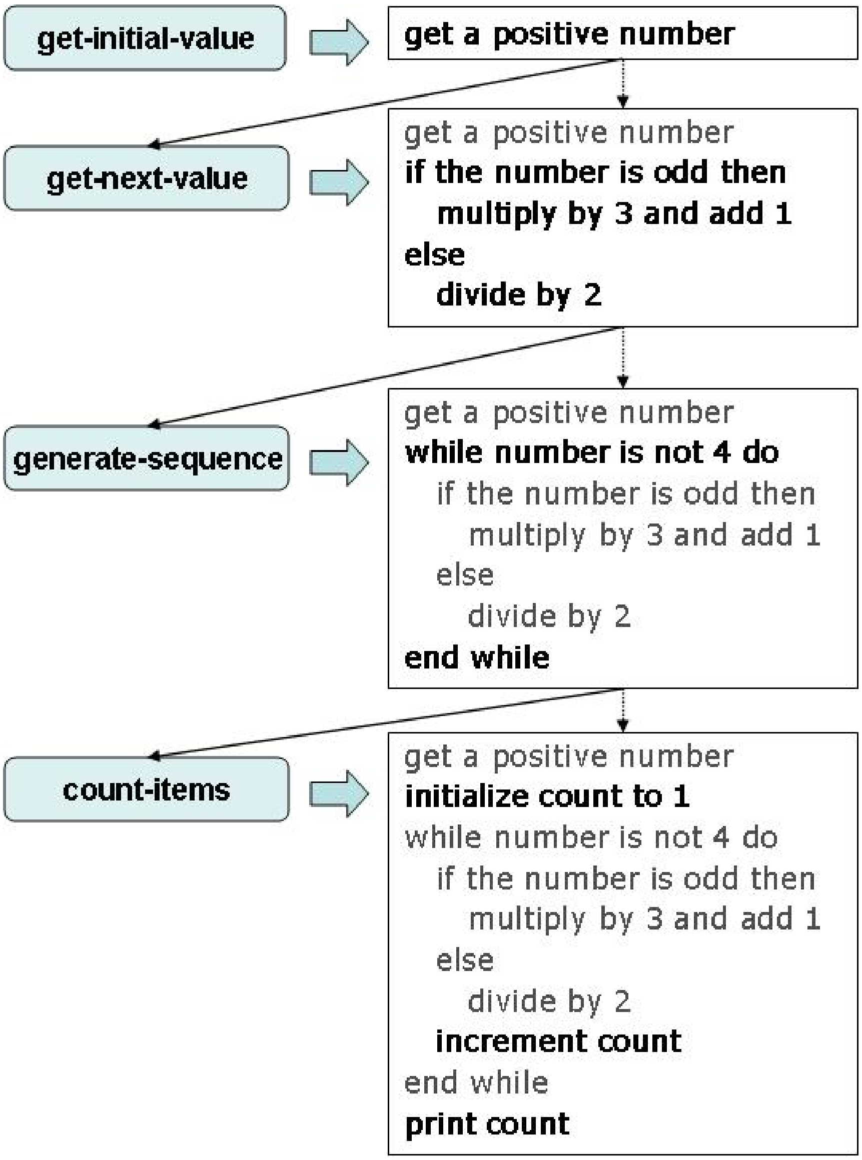
After achieving get-initial-value, most students prefer to follow the problem statement by applying the second and third steps (that involve checking for even or odd). This also follows

[8] When referring to a specific programming goal by name, a sans serif font is used.

the intuition from a hand calculation. The goal is therefore to get the next Hailstone value (get-next-value) and can be achieved by an update schema consisting of conditional statement and a appropriate assignment statements. After implementing the plan for this schema, it is simply *abutted* with the code that reads in the initial value (top part of figure 4.2). At this point, the program obtains the initial value and computes the one that comes after it. The next issue the student must confront is the "goto" step in the problem statement. Intuitively, the goal is to get yet *another* value; that is, to attempt next-value again. Some students suggest to implement another update schema (to abut with the one that already exists), but this approach fails to recognize the need for repetition. The correct goal to pursue next is therefore one to generate a full Hailstone sequence (generate-sequence). This can be done by wrapping a while loop around the existing conditional statement as shown in the bottom of the already existing code. This is shown in the middle of figure 4.2.

The only two goals remaining to complete the pseudocode come directly from the problem statement. The goals are to determine the length of the Hailstone sequence (count-items) and find the largest value in the sequence (find-largest). As with the other goals, the plans that achieve them can be easily folded into the existing pseudocode. Each requires a new variable, an initialization step, an update step inside the loop, and a print statement after the loop. Integration of each plan is similar, and because of space, figure 4.2 only shows integration of the counter plan.

The goal structure of Hailstone is fairly straightforward. The challenge for most novices, in my experience, comes in recognizing the two tacit goals: next-value and generate-sequence. These goals are not explicitly stated in the problem statement and thus are easier to overlook. They represent abstractions that can be inferred from the four steps described in the problem statement (steps 2 through 4, figure 4.1). Steps 2 and 3 can be viewed as one conceptual step: *to get the next value*. It could certainly be argued that this goal does not *need* to be recognized in order to implement a solution, and this is true. However, one of the primary pedagogical aims of CPP is to help students form the abstractions and view programs as a collection of plans. In this light, it is important to clearly establish such goals. The other 3 goals, get-initial-value, count-items, and find-largest, are all explicitly stated in the problem statement and thus much easier to recognize.



```
get a positive number
```

```
get a positive number
if the number is odd then
   multiply by 3 and add 1
else
   divide by 2
```

```
get a positive number
while number is not 4 do
   if the number is odd then
      multiply by 3 and add 1
   else
      divide by 2
end while
```

```
get a positive number
initialize count to 1
while number is not 4 do
   if the number is odd then
      multiply by 3 and add 1
   else
      divide by 2
   increment count
end while
print count
```

Figure 4.2: A staged solution to the Hailstone problem. Goals are shown along the left side and the pseudocode achieving those goals along the right. New steps for each goal are shown in boldface.

4.3 THE TUTORING MODEL: COACHED PROGRAM PLANNING

The staged solution to the Hailstone problem just presented above demonstrates a key part of the Coached Program Planning (CPP) model of of tutoring. Most novices will attempt to achieve most or all program goals at once, and subsequently struggle intensely with the interactions between the plans involved in the solution (Spohrer and Soloway 1985). CPP is designed to slow novices down to prevent a precipitant implentation. The intent is to provide the scaffolding novices need in order to plan effectively. This section begins with a discussion of the history behind CPP and continues on to discuss the details of collecting a corpus and how it was examined. From here, the basic tutoring model is laid out. This section sets the backdrop for presenting the tutoring tactics discussed in section 4.4.

4.3.1 Corpus analysis

In section 4.1.1, I described the lead-up to CPP. Based on some intuitive notions of what I believed novices needed, an environment for pseudocode planning was constructed and then pilot students were put through it with me as the tutor. In this section, this corpus is described in more detail, and discuss the process I followed to analyze and learn from it.

4.3.1.1 Corpus overview The full CPP corpus consists of 48 tutoring sessions across 5 problems, all performed between the summer of 2001 and the summer of 2003 with sections of introductory programming at the University of Pittsburgh (CS0007 and CS0401, in particular). The main reason for collecting the corpus was to (1) collect a large sampling of student utterances and (2) assess and understand a variety of tutoring tactics in the context of program planning. Sessions were offered for a variety of the programming assignments given during those semesters, and so the corpus covers several problems in addition to Hailstone and Rock, Paper, Scissors:

1. **Numbers to words**: input a number between 1 and 999 then display the number in English words.

2. **Weather calculations:** implement a menu-driven program that computes several popular weather-related formulas.
3. **Baseball statistics:** input raw baseball data and calculate derived statistics (such as slugging percentage) to store in a file.

Based on the awkwardness of the resulting dialogues for the latter two problems, it became clear that these problems were somehow not a good fit for CPP. The problems seemed to require less insight than the other programs and more "busy work" to implement them properly. The numbers program was a good fit, but overlapped too much with Hailstone and RPS to be given during the same semester with respect to its relationship with the syllabus. Hence, the analyses presented in this chapter focus primarily on Hailstone and RPS. Luckily, they constitute a large portion of the corpus.

Of the 48 dialogues, 14 covered the three problems listed above, and these were not used in this analysis. After this, 22 are from the Hailstone problem and 12 from RPS. In the spirit of experimentation, 3 of the Hailstone and 4 of the RPS sessions (done in the summer of 2003) were done *without pseudocode*. They were performed to see if students would respond to highly abstract planning dialogues which discussed goals and schemas only (with no programmatic reifications at all, just English text). Although there was some overlap with the full CPP dialogues, on the whole these were also too awkward to draw useful data from. For example, there never was any pseudocode of which to refer, and students who became lost early in the session gave very poor answers for the remainder of the time. The content of this chapter is therefore based on the 27 dialogues that involved full CPP over the Hailstone and RPS problems.

4.3.1.2 General procedure for corpus analysis

In analyzing the dialogues in the corpus, my goals were to find out how novices talked about programming (see chapter 5), and also to learn how student misconceptions would surface in dialogue. In addition, I also wanted to retrospectively analyze my own tutoring behavior. Although having outside tutors may have been beneficial in claiming generality regarding these results, my aim was to build an automated system, not to seek general truths about the behaviors of tutors. In many ways, it was an advantage to have been the tutor because I was able to easily explain my own tutoring behaviors, even when they were not obvious from the textual content alone. For example, the decision to use more involved tutoring tactics at certain points in the session were often influenced by my incoming knowledge of that particular student.

To break down a tutoring session, the first step was to find the locations of the top-level what questions (i.e., goal-eliciting). These were easy to locate since all tutoring sessions followed a staged approach to solving the problems in pseudocode. This allowed easy collection of the answers to these questions. Next, within each of these sections, the corresponding how-questions were identified, and their initial responses collected. The pattern seemed to generally hold, except when:

- The student provided a schema-oriented answer to a goal question.
- The student gave a prematurely correct answer, to which the tutor would, in place of re-asking, just remind the student later of that answer.
- The goal question was so simple, that the goal and how to achieve it were nearly identical (e.g., ”read in a value”).

With the lines drawn between the discourse segments, I was able to look in between them to find examples of tutoring tactics.

Before identifying tactics, it was first necessary to understand the different kinds of answers students were giving. The quality of the answers, not surprisingly, seemed to play a major role in determining the tutorial responses. For example, an overly-specific answer is often responded to with an *elevation* tactic that tries to bring the student up to thinking about goals (these are discussed in section 4.4). After knowing what kinds of answers are given, the next step was to look at the tutorial responses in close detail. In identifying tactics, I began to organize them in different ways. The top-level split that worked best for me was between simple, 1- or 2-turn tactics, and those that required multiple turns to complete. I found that the longer remedial dialogues almost always involved a reference to some example, either in retrospect or with a brand new example (this is discussed later in section 4.4.2.4). Finally, with the tactics delineated in the dialogues, it was easier to convert them into dialogue knowledge sources (by hand).

4.3.2 General aims and principles

Through reflective-practice and introspection on this first part of the CPP corpus (before the details had been fleshed out), I was able to solidify some of the aspects of the underlying pedagogy. These are now presented. In a CPP tutoring session, the student and tutor collaborate to build a natural-language-style pseudocode solution to a problem. Ideally, the student has already read the problem statement, but has not yet attempted an implementation. Dialogue is the vehicle for this collaboration. In the last section, the underlying decisions behind CPP were discussed. These were: adopt early (pre-practice) intervention, natural language dialogue, pseudocode, and staged design. Moving on now to the *role* that CPP should play and the principles that guide tutorial interaction.

Principle #1: Make planning part of learning about the problem.

The natural tendency of most novices is to enter prematurely into an implementation phase, thereby skipping any meaningful planning activities (see section 2.2.2). As discussed, the travails of an unprepared novice with an editor and compiler are often significant. The approach taken in CPP is to integrate planning with the process of reading and learning about the problem. That is, the aim is to traditional methods of assignment dissemination (e.g., a printout) with an interactive experience that will (hopefully) better prepare them for an independent implementation phase. The long term hope is that novices will begin to understand that pre-planning is an important part of real software design. Unfortunately, the scope of the experiments in this dissertation (chapter 7) do not allow this outcome to be addressed conclusively.

Principle #2: Model and support the cognitive planning activities that novices are known to underestimate or even bypass altogether.

One explanation for why novices fail to plan is simply that they do not know how to do so. One of the overarching goals, then, of tutoring in CPP is to scaffold program planning activities and when necessary, demonstrate them. In the CPP model, this consists of decomposing the problem, identifying the schemata and plans necessary, and composing them in the form of pseudocode. The basic motivation is to help the student understand the problem and develop an idea of how to solve it without the distractions presented by programming language specific issues. Even if the student is not able to answer correctly very often, the experience will have at least exposed some of the subtleties of the problem.

Principle #3: Exploit commonsense understanding of the problem to help novices graduate to a more algorithmic understanding.

Knowledge-lean tasks are easy to understand and simulate by hand (see section 4.2.1. This principle implies that tutoring should take full advantage of this fact whenever possible to help students make problem solving and design suggestions. The idea is to use what the student knows and understands, and help connect it to what they do not yet understand, i.e., how to program the same task. This principle suggests that the tutor should use the natural ability of a student to perform a hand calculation and to subsequently scaffold the *decompilation* of the steps. This can then be used to help the student make the necessary abstractions and to produce a program.

Several reasons to use natural language tutoring were provided in section 4.1.2.2. The argument was that although some aspects of natural language do present extra challenges, it remains a viable medium in which to learn about programming. A programming system that uses natural language also affords another way for students to express their ideas. For example, it is possible that a student who cannot express an idea they have in the strict syntax of a programming language *could* do so in their own words. In sum, *the idea is to exploit natural language and common sense to make the implicit knowledge of programming more readily available to novices.* Natural language allows us to leverage students' commonsense problem solving and programming knowledge to elicit problem decompositions and to support the application of these general skills within the context of programming.

Principle #4: Use the low-level tendencies of novices to help them build abstractions and think at the level of schemata.

A broad aim of CPP is to be aware of the "code-first" tendency of novices while gently and indirectly helping them to think more abstractly about the problem. Novices lack problem solving ability: given a problem statement, students are *not* likely to have the ability articulate a clear set of programming goals. Novices tend to think of programming "bottom-up" (Rist 1989; Soloway 1989; Wender, Weber, and Waloszek 1987), and thus are likely to have the ability to suggest some applicable lower level programming concepts (e.g., "we need an if statement"). The basic approach adopted in CPP is to *take advantage of natural novice tendencies and use them to steer novices into to a more expert-like, abstract view*. The lower-level elements of programming are important, and novices should receive positive feedback when these suggestions are correct. However, they need help *elevating* their view, realizing that dispersed lines of code in a program are often best viewed as one, and understanding which lines of code accomplish which goals.

The fact that novices are generally able to produce the correct program parts, but fail to bring them together effectively (Spohrer and Soloway 1985), is powerful evidence that more support is needed during the planning phase of programming. Novices *can* identify the lower level necessities, but just need guidance to connect these up with the more abstract notions of problem solving and design. This is why early, pre-practice intervention is important and why CPP is designed in this fashion.

4.3.3 3-step pattern

In section 4.1.2.4 a staged model of program development was presented. The heart of this approach is that identification and achievement of goals are interspersed. That is, once a goal is identified, it is implemented before another goal is another goal is identified. This leads to an "evolving" solution that simplifies the plan merging aspect of solving the composition problem (Spohrer and Soloway 1985). The staged approach produces a 3-step dialogue pattern consisting of the identification of goals, how to achieve them, and actually achieving them (figure 4.3).

Since the tutor's overarching goal is to elicit the design from the student, each step in the pattern corresponds to a tutor question along with, perhaps, a sub-dialogue aimed at eliciting the correct answer. Although there are some exceptions in the full corpus (described in section 4.3.1.2), this pattern seems to be consistent throughout the corpus. It reveals a

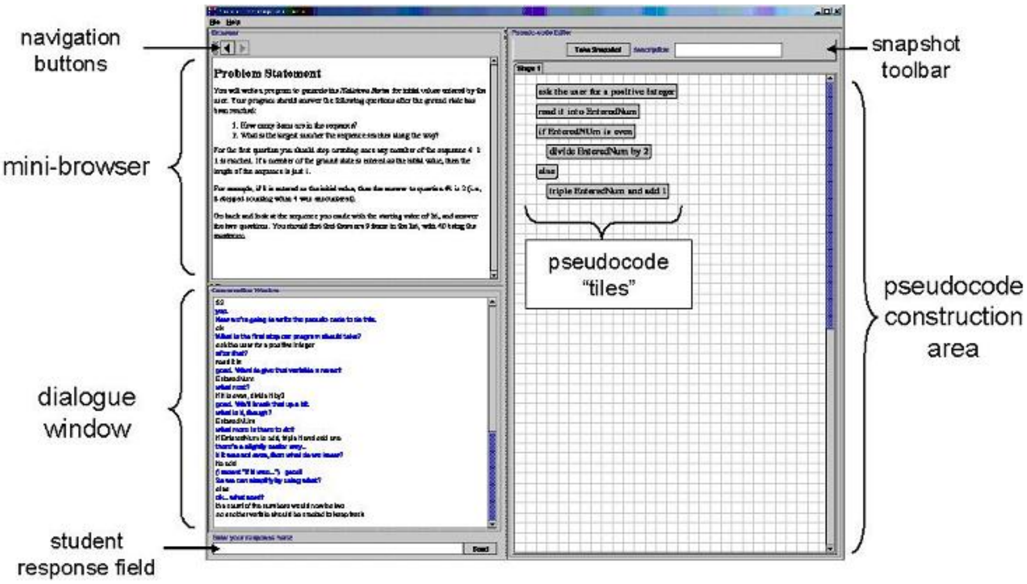
1. identify a programming **goal**
2. identify a **schema** for attaining this goal
3. **realize** the resulting **plan** in pseudocode
   a. create pseudocode step(s) achieving the goal
   b. place the steps within the pseudocode

Figure 4.3: 3-step pattern resulting from the use of staged-approach to design.

progression from goals, to schemata, and ultimately to plans which are then integrated into the pseudocode. When represented as natural language utterances, schemata and plans take on slightly different forms. For example, a schema can be identified by vague utterances such as "keep a counter" whereas plans are more precise and require the mention of program-specific variables: "initialize counter to 0 and then increment it inside the loop." There is pedagogical value in recognizing this distinction between schematic forms: helping novices to identify general strategies first and *then* transition to the more precise representation of plans supports the more abstract view promoted by CPP.

4.3.4 Elements of a tutoring session

The first task a student is directed to do in a CPP tutoring session is to read the problem statement. Upon completion, the tutor and student then perform an an interactive hand calculation that is used to confirm the student's understanding of the target task and to act as a demonstration the desired program behavior. It also provides an example to which the tutor may refer to later. Once the student completes the hand calculation (as stated earlier, it is usually quite easy to do), the dialogue shifts to writing pseudocode, which plays the role of the artifact in the design. Following the 3-step pattern, the tutor repeatedly asks the student to suggest goals, give ideas on how to achieve them, to identify which steps to add to the pseudocode, and to determine where they belong. The pattern is repeated until



Problem Statement

You will write a program to generate the Hailstone Series for initial values entered by the user. Your program should answer the following questions after the ground state has been reached:

1. How many items are in the sequence?
2. What is the largest number the sequence reaches along the way?

For the first question you should stop counting once any member of the sequence 4 2 1 is reached. If a member of the ground state is entered as the initial value, then the length of the sequence is just 1.

For example, if 8 is entered as the initial value, then the answer to question #1 is 2 (i.e., 8 stopped counting when 4 was encountered).

Go back and look at the sequence you made with the starting value of 26, and answer the two questions. You should find that there are 9 items in the list, with 40 being the maximum.

52
yes.
Now we're going to write the pseudo code to do this.
ok
What is the first step our program should take?
ask the user for a positive integer
after that?
read it in
good. What do give that variable a name?
EnteredNum
what next?
if it is even, divide it by 2
good. We'll break that up a bit.
what is it, though?
EnteredNum
what more is there to do?
if EnteredNum is odd, triple it and add one.
there's a slightly easier way..
if it was not even, then what do we know?
its odd
(I meant "if it was...") good!
So we can simplify by using what?
else
ok... what next?
the count of the numbers would now be two
so another variable should be created to keep track

```
ask the user for a positive integer
read it into EnteredNum
if EnteredNUm is even
  divide EnteredNum by 2
else
  triple EnteredNum and add 1
```

Figure 4.4: CPP Environment

all programming goals have been satisfied, and the pseudocode is complete. It should be noted that the dialogue does *not* typically include the reification of schemata into complete plans – this happens with creation of pseudocode steps. It is a blurry line, however; novices tend towards the concrete, and so it often is easier to talk about existing program constructs and variables from the pseudocode. In these cases, the tutor will commonly engage an *elevation* tactic to make sure the student is able to link up the lower-level pseudocode with the higher-level, problem solving issues (see section 4.4).

Although it would be feasible to perform CPP in a face-to-face context, the ultimate goal of this research was to produce an automated system to perform it (i.e., ProPL, chapter 6). The corpus was therefore collected via computer-mediated tutoring sessions over a network. The environment used is shown in figure 4.4. The interface consists of three main windows:

1. A **mini-browser** (upper left) that displays HTML pages and remains available throughout the tutoring session.

2. A **dialogue window** (lower left) that allows communication between the tutor and student. Student contributions are made via the text field below the window.
3. Finally, the **pseudocode window** (right half), which contains draggable *tiles* containing pseudocode text, each of which represents a step in the solution.

After using the mini-browser to read the problem statement and doing a hand calculation in the dialogue window with the tutor, the student and tutor work together to build the solution in the pseudocode window. As steps are suggested by the student, they are either rejected, accepted with some modification, or fully accepted by the tutor. After this, the tutor creates tiles, and the student drags them into the pseudocode area. Not allowing the student to create tiles directly puts the tutor in a "filter-like" role, preventing wrong solutions from being created. Instead, these poor ideas are caught in the dialogue and remediated there instead of within the program. The content of the tiles (i.e., the text of the pseudocode step) is kept as close to the student's actual language as possible at the tutor's discretion. Upon completion, the tile outlines are removed leaving the pseudocode in a more traditional form for the student to print out and use during implementation.

During a CPP dialogue, the tutor acts as a filter for the student's ideas and so the resulting pseudocode is guaranteed to be correct. Thus, heading into implementation, students are armed with a better understanding of the problem and a better idea of the overall algorithm. The individual pseudocode steps are implicitly approved and often edited by the tutor before making their way to the solution. Thus, the style of the pseudocode is up to the tutor. The basic requirement is that the student should require minimal (if any) training to generate the text for the steps. In summary, each step is suggested by the student, created by the tutor, and finally placed by the student.

To reify the staged approach to design in the experiment, the environment also supports *snapshots* of the pseudocode at different times during its development. Basically, after a goal is identified and implemented, the student is asked to summarize what it accomplishes, and then a snapshot is taken (by clicking the button in the upper right corner of the interface). This "freezes" that version of the pseudocode so the student can remember what the code looked like at that moment. The CPP environment then produces a new copy of the tiled window that allows the student and tutor to continue working on the solution. At the end of the session, the student is allowed to review all of the stages by clicking on the tabs. The snapshot capability was dropped in ProPl in lieu of a list of programming goals with descriptions (these are called *design notes* and are described later in section 6.1.3).

4.4 ELICITATION TACTICS

Good tutors are stingy when it comes to giving answers – they prefer to *hear* them. In section 3.1, the point was made that learning is enhanced when the student plays a greater role in the tutoring; the student should be responsible for as much of the problem solving as possible. This chapter turns now to a critical component of of CPP: how to elicit answers from students. Two categories of elicitation tactics are identified: top-level questions and remediation.

4.4.1 Top-level questions

As indicated earlier, the 3-step pattern (figure 4.3, page 57) dictates the initiating questions used by the tutor to elicit design choices from the student. Each part of the 3-step pattern corresponds to a question intended to elicit that item from the student. To elicit programming goals from the student, questions such as the following are used:

- "What is the first thing the program should do?"
- "What should we work on now?"

To support transition between goals, sometimes it is a good idea to state what was just achieved before posing the question:

- "We've got the initial value. What do you think we should work on next?"
- "With both choices now loaded, what are we now ready to do?"
- "At this point, the program plays a game. What should it do next?"

Setting up the question in this way also helps to ground the state of the solution in the dialogue. The student is reminded of where the solution stands and of the purpose of work just completed. Finally, it also contextualizes the question so that the connection between the hand simulation and the program under construction is easier to make.

Once a goal is established, the next step is to decide how to achieve that goal (step 2 of the 3-step pattern). The tutor's objective is to elicit a schema suggestion from the student by posing a "how" question. Some typical examples are:

- "How do you think we can do that?"
- "Any ideas on how we can update the pseudocode to accomplish this?"
- "How should we proceed with that?"
- "How do you think we can get the program to count the items?"

Such questions immediately follow the establishment of a goal in the dialogue, and so it is less common to preface them as sometimes done with goals. In cases when establishing a programming goal is difficult for the student, however, it is not uncommon to see the "how" question posed in a more precise way (like the final bulleted example above).

4.4.1.1 A comment about top-level questions Although top-level questions seem to the student to be open-ended (e.g., "What should we work on now?", sometimes in reality they are not. For example, in many cases a goal-eliciting question is actually a closed-answer question when considering the set of correct responses. This is most true when the expectation for a goal-question is an explicitly stated goal in the problem statement. For tacit goals there is a slightly greater range of expression since there is no vocabulary established in the problem statement, yet it is still really a closed-answer question that appears like an open-ended one.

The schema-eliciting questions (i.e., "how" questions) are, however, closer to being truly open-ended. Students have a far greater range of potential answers available to them. For example, a response can be a good description of the schema, the key step in the schema (or plan), a programming primitive, and so on. Basically, anything that can be interpreted as related to or part of the appropriate schema can be considered correct. In chapter 6 we pick up this discussion to explain how ProPl handles this wide variety of potential answers. In general, the issue of open- and closed-answer questions is a difficult to classify.

For the purposes of this research, the only suggestion made is that it is important to *confront* students with seemingly open questions to encourage them to think through alternatives and organize their planning-oriented knowledge.

4.4.1.2 Categorizing student answers Obviously, the quality of a student's contributions is extremely important in determining how to respond. A classification scheme similar to the one below has been developed as part of the CIRCSIM-Tutor (Glass 2001). Aside from being completely correct or incorrect, students can be wrong (or right) in a variety of ways:

- **partially correct/incomplete**: covers only a portion of the expected answer.
- **partially incorrect**: contains some aspect that is incorrect, but also something correct (else it would be completely incorrect).
- **overly vague**: captures the gist of an expected answer, but is too imprecise in some important way(s).
- **overly specific**: too much detail is given when a more abstract or general response is expected.
- **correct, but premature**: true, but not timely; usually indicates the student is overlooking something of greater immediate importance.

The initial response to the 3-step questions can reveal a great deal about the student's ability and appropriate tutoring tactics for later in the dialogue. Consistently poor answers that indicate little understand portend more example-based tactics and more explicit lines of questioning in the corpus. Similarly, strings of good answers early on seem to establish the student's competence, and lead to more abstract tactics by the tutor.

4.4.2 Remedial tactics

The more successful a student is answering top-level questions, the faster progress is made developing a solution in the dialogue. This holds for the simple fact that the student begins closer to (or at) the correct answer. This is no guarantee that the student fully understands, of course. It simply means the path to finding a good answer is shorter (in the dialogue). When there is a problem with the student's answer, as there often is in the corpus, the tutor

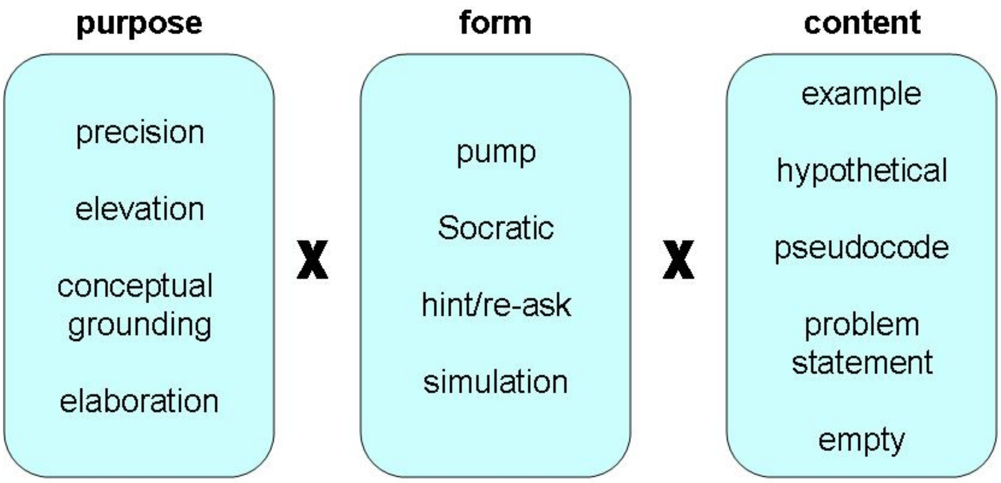


Figure 4.5: Three dimensions of tutoring tactics in CPP. The *Purpose* describes what kind of answer the tactic is intended to elicit, *form* indicates the structure of the interaction, and *content* simply reveals what the form refers to. The columns are orthogonal and each tutoring tactic falls into some category along each dimension.

must take remedial action to elicit a correct response. Sometimes the response is simple and direct, while in others it is somewhat involved (even tedious). In this section, a number of representative remedial tactics are shown and a framework from which to understand their use is explained.

4.4.2.1 Purpose, form, and content To describe the remedial tactics of CPP, it is useful to analyze them along 3 dimensions: *purpose*, *form*, and *content*. A tutorial purpose is simply a pedagogically driven intent of the tutor largely motivated by the quality and classification of the student's initial answers. The form of a tactic is its structural make-up or how it takes shape in the dialogue. Finally, the content of a tactic is that which is referred to by the form. An overview of the various values each dimension can take on is shown in table 4.4.2.1.

The purpose of a CPP tactic is most heavily influenced by the classification of an answer, be it too vague, or partially incorrect, etc. Generally, the tutor has an *expectation* of a correct answer and at what level of abstraction the answer should be cast. Also, sometimes the tutor may need to confirm understanding of something in the dialogue or simply to tell the student a missing piece of information. All tactics in CPP therefore serve one of the following purposes:

1. **precision**: elicit a more specific answer from the student.
2. **elevation**: elicit a more general or abstract answer from the student.
3. **conceptual grounding**: confirm the student's understanding of how something relates to the problem or pseudocode.
4. **elaboration**: (non-interactive) direct explanation to the student, without another question.

The quality of the student's answer directly determines the purpose adopted by the tutor. For example, if a student's answer is too vague, a precision eliciting tactic is chosen. Similarly, if a student provides unnecessary or spurious detail, then the tutor attempts to *elevate* the students thinking to a more abstract level. Elevation can be used to either help the student think in terms of goals and schemata, or to make an important generalization specific to the problem from a concrete example. Several examples of these tactics are shown later in this section.

Once a purpose is in place, the tutor must decide both form and content of the response; these are shown in the second and third columns of table 4.4.2.1. The various forms used in a CPP tactic are:

1. **pump**: a simple prompt asking for more information, such as "Can you say anything more about that?"
2. **hint & re-ask**: provision of some requisite information the tutor believes the student does not have, followed by a re-asking of the question.
3. **Socratic**: an extended dialogue consisting of a string of tutor questions intended to culminate in an "ah-ha" moment for the student as the answers are digested and related to one another.
4. **simulation**: another extended technique involving either a *hand calculation* or simulated execution of the pseudocode.

Pumping, a tactic used frequently used by AutoTutor (Graesser, Person, and Harter 2001), and hinting, used by just about every ITS, are simple, one- or two-turn tactics. They are most useful when there is a sense that the student is on the cusp of the right answer and only needs a slight nudge. If frustration is obvious or if the student's answer is far from correct, however, a simple tactic is likely to fail and a more involved, multi-turn tactic is called for. In the CPP corpus, such tactics fall under the two broad categories of *Socratic* and *simulation*. Socratic tutoring, pioneered in the ITS community by the WHY system (Stevens and Collins 1977), is usually triggered by a flawed answer indicative of a deeper misunderstanding.[9] Finally, simulation-based tactics are quite common in the corpus. It is widely accepted that students in general respond positively to concrete examples, and so this is no surprise.

The third and final dimension of tutoring tactic framework is content. The various sources of content for tactics used in the CPP corpus are:

1. **examples**: a concrete example of the target task.
2. **hypothetical**: an imagined example or reference to a previous example, sans the details.
3. **pseudocode**: the pseudocode solution.
4. **problem statement**: the description of the target task, program requirements, and often an example to read.
5. **empty**: no information content (only applies to a pump).

Unless the context of a tutorial response establishes something to refer to (such as pumping), it is necessary for the tactic to "set the stage" for eliciting the desired piece of information. Simply put, the content of a tutorial tactic gives the student something to think about and work from while generating a response.

The rest of this section is devoted to examples of these tactics in action taken from both the human-human study of CPP. Taking the cross product of the 3 dimensions discussed above, 80 different categories of tactics are possible (4 purposes, 4 forms, and 5 kinds of content). However, many of these are not used because they are incoherent. Two examples

[9] Strictly speaking, a Socratic dialogue involves a series of questions all based on a faulty assumption by the student. At the end of the dialogue, it is hoped the student will come to recognize the error. However, over time the definition has broadened to refer to any dialogue that consists of repeated questioning.

1 T What should we work on now?
2 S **I don’t know**
3 T You might want to take a look back at the problem statement.
4 T Give it another shot. What should we work on next?
5 S **count the items in a sequence**
6 T Good job.

Figure 4.6: A goal-eliciting KCD that uses the tactic of pointing to the problem statement (generated by ProPl).

are pumps with non-empty content and elaboration with follow-up questions. Nonetheless, there is still a rich set of tactics to discuss and so the next section begins this discussion with goals. The examples below were selected to give a broad flavor of the different tactics used in the corpus.

4.4.2.2 Goal remediation tactics Identifying program goals is often quite easy to do. When goals are explicitly stated in the problem statement, such as count-items and find-largest in the Hailstone problem, students in the corpus are nearly always able to provide the answer with little difficulty. In the few cases when a student is not able to identify an explicit goal, a gentle nudge back to the problem statement is usually all that is necessary. An example appears in figure 4.6. In this case, the student claims not to know what to do next in line 2, but with the suggestion to look at the problem statement in line 3, is able to identify the correct goal.

To analyze this tactic in terms of purpose, form, and content, it is first necessary to understand the student’s answer. Since it is completely void of domain-related content, in this context “I don’t know” is considered extremely vague. Thus, the tactic is classified as follows:

- The purpose is to elicit more **precision** in the answer.
- The form is to give a **hint** and then **re-ask**.
- Finally, the content comes from the **problem statement**.

1 T What do we need to work on next?
2 S **Finding the how many numbers are in the sequence**
3 T are you sure? [pause]
4 T does our program generate an entire sequence?
5 S **We need to see if the new number is now odd or positive.**
6 T right... and after we do that, are we done?
7 S **Not unless we reach the ground state.**
8 T how can that be determined in the program?
9 S **by a loop**
10 T right!

Figure 4.7: An attempt to elicit the generate-sequence goal of Hailstone turns into a schema-eliciting dialogue (this is a human-human example).

When successful, simple tactics such as this one allow the dialogue to move forward quickly since they can be resolved quickly and rarely require any subsequent deep elaboration.

When a goal is tacit, however, it proves to be much more of a challenge to elicit. In these cases, an inference or generalization is usually required to properly identify the goal. To illustrate the difficulty of eliciting such a goal, figure 4.7 shows a dialogue of a human tutor attempting to elicit the implied generate-sequence goal of Hailstone. Two tactics are present in this figure. First, the student is ready to count the items in the sequence, however, there is no sequence yet to count. The tutor attempts to *pump* the student for the information in line 3, but when this fails, engages in a slightly more involved tactic.

The tutor asks about the state of the pseudocode in line 4, thus engaging a a different tactic. This classification for it is as follows:

- The purpose is **elevation**, since the tutor is seeking an abstract answer.
- The form is **Socratic**.
- And the content is from the **pseudocode**.

Even though the dialogue appears good at first glance, the tactic is not technically successful: the goal the tutor accepts is to implement a loop (line 9). This is markedly different from saying something like "we need the program to produce a sequence" since it involves a programming construct. Goals in CPP are expected to be more abstract entities that should stand semantically closer to problem statement rather than a programming concept. Second, the tutor seems to blur the line between goals and schemata by encouraging the student to think about how to update the code (line 8). The 3-step pattern is not specifically followed since the tutor seems to switch from looking for a programming goal to accepting a technique instead. In ProPl, this student would have received positive feedback for responses such as those in the dialogue, but the more abstract, problem-derived goal would have been discussed.[10] There are several other aspects of this example that are important to note:

- The student takes initiative by ignoring the tutor's precise question in line 4 (it is a yes/no question).
- The answer given by the student in line 5 is part of the plan that achieves next-value, which in turn is a component of the generate-sequence plan. This reasoning confirms that the student is "in the ballpark."
- Similarly, in line 7 the student mentions "the ground state", which is part of the plan to achieve generate-sequence.

In this dialogue, the student seems to lack a heuristic of the form *if a sequence of values needs to be generated, use a loop.*[11] The observation that the student highlights the plan parts before thinking about the whole plan *as a unit* stands in harmony with findings discussed in chapter 2 that described the low-level perspective novices tend to adopt. That is, novices tend to view programs "line-by-line" and prefer a depth-first search approach to programming. This is, of course, at odds with the breadth-first, "top-down" approach commonly taught in introductory programming courses.

There is one last thing to say about the dialogue in figure 4.7.[12] The student accidentally says "positive" instead of "even" in line 5. The slip is overlooked by the tutor in this case because of the very high probability the student meant to say "even." In the corpus, close to 40% of the students in the CPP corpus make a positive/negative for even/odd slip at

[10] This was one of the earliest dialogues collected during the human-human study, and so the tutoring was still at an experimental stage.

[11] Spohrer (1992, p.19) refers to this as an *expand* plan that inputs a single value and produces a stream of values.

[12] The audience moans.

1 T So, what is the first step in playing a game of RPS?
2 S **knowing the rules**
3 T Well... If we imagine a real game between two people, each player begins by slamming their fist into their other hand three times. What are they doing on the third slam?
4 S **choosing rock paper or scissors**
5 T Correct. So, the game starts with each player making a game choice.

Figure 4.8: Eliciting the first step of a game of Rock-Paper-Scissors using a hypothetical situation (this is a ProPl dialogue).

some point. When asked about it, students nearly always immediately repaired the mistake, saying they intended even or odd. Although a mild slip, it occurs far too frequently to not mention here. Thanks to these observations, ProPl is able to detect this slip and make the proper assumption about what was meant. The response is simply to elaborate on the difference, and then assume the property of evenness (or oddness) was originally intended.

The next example of a goal-eliciting tactic uses a hypothetical situation and is shown in figure 4.8. This is best classified as an elevation tactic because the student suggests a goal to inform the user of the rules of the game (line 2). While a good idea, this is more "busy work" than anything else. The purpose here is to elicit the more important goal of getting the player inputs for a game of RPS. The hypothetical game is used to help the student take advantage of a commonsense understanding of how to play. The tutor refers to the act of "slamming their fist," which is intended to help the student visualize the act of make a selection. The general strategy being used by the tutor in this example is to *decompile existing knowledge* and observe the steps. In this case it is used to elicit a goal, but it is most commonly used to elicit schema components in the context of a more elaborate hand simulation.

The next dialogue is another example of an elevation tactic. In this example (shown in figure 4.9), the tutor asks the student to consider why a loop was appropriate. In a way, this is the kind of dialogue that the human tutor *should* have pursued in figure 4.7. When asked why a loop is needed, the student provides a fairly typical response from a novice in line

1 T ...what do you think we need to do?
2 S **create a loop**
3 T That's right. We will have to add a loop.
4 T Let's figure out why. What made you think of using a loop?
5 S **because a loop continuously cycles until a desired number is reached.**
6 T A loop is definitely needed, but it is important to think clearly about what purpose it serves. In this case, a loop will allow you to create a whole sequence of numbers.

Figure 4.9: Example of an elevation tactic to elicit a programming goal. The tactic is to simply ask why a particular technique is needed (ProPl dialogue).

5. Basically, the student is "stuck" at the line-by-line, more concrete level. Although the tutor does give up a bit too quickly (ProPl is tightly bound by its dialogue capabilities), the ideal abstraction is stated by the tutor and (hopefully) understood by the student. In many cases, this is actually deemed "obvious" by students, even when they are unable (or perhaps, unwilling) to offer it as an answer themselves.

4.4.2.3 Schema/plan remediation tactics Schemata and plans are more complex than goals, and thus are harder to elicit from students. They consist of *components*, each of which can correspond to a line in a program, or another schema. Secondly, when asked a top-level "how" question, the range of responses that can be considered correct is quite large. For example, the answer can be a nice abstraction of the general approach represented by the schema. It might also simply be the first step of the schema. Most schemata are complex enough that it not feasible to expect a complete answer all at once, and so the general approach used in CPP is to use the student's initial response as a kind of test to determine the subsequent tactic. If the initial answer is on track and correct, then more open-ended questions can follow. If the student says "I don't know" or produces a weak response (such as "write a function"), then an elicitation tactic must be used. In this section, examples falling under each of the tutorial purposes are shown.

1 T Let's look at an example to find out how counting

2 might work.

3 T So, starting with 10. What is the count so far?

4 S **10**

5 T Well, we just have 10. The count is therefore just 1 now.

6 T After 10 in the sequence is 5. What is the count now?

7 S **2**

8 T Good job. And after 5 comes 16. What is the count now?

9 S **3**

10 T Right. So what is happening to the count?

11 S **it increments by one each time the loop runs a cycle**

12 T Yep. How often do we need to do this?

13 S **until the counter equals 4**

14 T We will need to add one to the count for each value we see.

Figure 4.10: Teaching an "on-the-fly" counting schema through the use of a hand calculation (with a human tutor).

Examples of elevation tactics

The first example builds off of the notion of knowledge decompilation described earlier. For knowledge-lean problems it is almost always very simple to do a hand calculation, and so in the tactic shown in figure 4.10, it is easy for the student to contribute in productive ways. The tactic can be classified as having the purpose of elevation, the form of a simulation, and the content of an example.[13] There are several important points to make:

- The tutor asks a series of very easy questions to answer (lines 3 through 9). This keeps the student involved and contributing.
- The natural way to count (on paper) is to look back over the sequence and do it *after* the sequence is done. This tactic introduces the student to a *dynamic* counting strategy, which is more suitable for use in a program.
- The sense of elevation in this tactic is to help the student transition from counting in an example to a general policy of incrementing the counter each time through the loop. The student provides the necessary generalization in line 11.

[13] A similar example was presented earlier in figures 1.1 and 1.2 in chapter 1.

1 T So, let's generalize. At the beginning of that step, you
compared the largest value so far (6) to what?
2 S **10**
3 T Right, that is correct. But let's try to say it in general.
It sounds strange, but what value did 10 represent at the time of
the comparison?
4 S **a new hailstone value**
5 T Good job. To summarize, then, we need to ask ourselves if the current
value is bigger than the largest so far. If true, we have a new largest.

Figure 4.11: Eliciting the comparison to find the largest value from a concrete example. In prior dialogue, the student had gone through the sequence 6, 3, and now 10, being asked to track the largest along the way (dialogue performed by ProPl).

- No pseudocode is mentioned - the discussion is purely at the schema level; this dialogue sets up development and integration of a counting plan for the pseudocode.

Concrete examples play an important role in the CPP corpus. They provide an easily accessible context in which to introduce the tacit knowledge of programming. In this case, the student is learning a counter schema. The student is gently transitioned from an intuitive simulation to a more algorithmic view. This discussion resumes later in section 4.4.2.4.

The second example of elevation involves helping the student come up with the correct test condition for finding the largest value in Hailstone. Novices generally struggle to understand which two values need to compared in the loop. For some reason, many students will suggest that the largest value be compared to the *initial* value of the sequence. Although this does happen in some hand calculations, it is certainly not part of the general solution. An example of a tactic that tries to elicit this abstraction (i.e., which two values to compare) is shown in figure 4.11. This tactic is best classified as elevation-hint/re-ask-example.

Not showing in the figure is the setup of the comparison question. In that setup, an example sequence was introduce that started at 6. The tutor tells the student that 6 is the largest so far (also an indirect hint that the largest variable should be initialized to the initial value of the sequence). After the first time through the loop, 6 remains as the largest because it is greater than 3. However, when 10 is encountered, the time comes to update the largest value. This is when the tutor asks the student to describe what 10 represents at that very moment (lines 1 and 3). The student here is able to answer correctly (line 4), however some students are confused by this question.[14] To sum up, the goal of the tactic is to help the student generalize from the easy to recognize fact that $10 > 6$ to the conclusion that the current hailstone value needs to be compared to the current *largest* value. Of course, there is a bit more work to be done, but this is the key abstraction targeted by this tactic.

Examples of precision tactics

Moving now to elicitation of plans, the tutorial purposes almost always fall under the category of precision. This association makes perfect sense: plans are the instantiated versions of schemata, and thus involve specific details that will soon be integrated into the pseudocode solution. The dialogue showing in figure 4.12 demonstrates such a tactic. In this example, the tutor is helping the student hammer out the termination condition on the main Hailstone loop. The student makes a poor suggestion for the loop termination condition in line 2 by suggesting that the loop stop when an even number is encountered. This indicates that the student has made a *merging* error by confusing the plan for generate-sequence with the next-value plan. The only time even and odd matter is in the calculation of the next value.

To fix this mistake, the tutor uses a hypothetical example in line 4 and with the student responding with something a bit more coherent ("the same numbers" refers to the 4-2-1 repeating ground state sequence of Hailstone), the student is able to enunciate the perfect condition (line 8). The full classification of this tactic is therefore precision-simulate-hypothetical.

The final example comes from the RPS problem. This program is the student's first opportunity at implementing functions (methods). A good portion of their tutoring in CPP is therefore about the interface, requirements, and usage of these functions. As mentioned earlier, students are able to determine the winner of a game with little thought. However, to write a function to compare two choices and return the winner is an extremely daunting task. The dialogue in figure 4.13 shows the tutor helping the student establish the background

[14] Some of the stranger responses in the ProPl corpus to the question posed in line 3 are "three times three plus one," "even value," and simply "3."

1 T What condition will we need on that loop?
2 S **while hailnumber is even do**
3 T think about that. you want to repeat the even/odd check over and over.
4 T in the example you did, when did you stop?
5 S **when it went to the same numbers**
6 T yep. that was called the "ground state".
7 T so we'll loop while what is true?
8 S **hailnumber is not equal to 4**
9 T good.

Figure 4.12: Eliciting the loop termination condition for the Hailstone problem. The student exhibits a *merging* error by suggesting that condition should look to see if the number is even (dialogue is with a human tutor).

necessary to implement a function that determines the winner of a game. The dimensions of this strategy are **conceptual grounding** (purpose), **Socratic** (form), and **hypothetical** (content).

Although ProPl is slightly too didactic in this example, the student seems to understand the conclusion. The correct answer (that there are 9 combinations) is given in line 10. Many students in the corpus answer 6 to the question posed in line 9, even with the elaborate setup. After this dialogue, the student is ready to talk about the conditional statements used in the calculation of a game winner knowing that there are 9 cases to handle.

4.4.2.4 More on the use of examples

In general, the use of concrete examples in tutoring is known to be a successful technique. Examples can be used, for instance, to directly confront misconceptions (Brown 1992), to provide a concrete reference for an abstract concept or skill (Heffernan 2001), or, in the case of programming, to act as data on which the program should run (Reiser et al. 1992). Although precisely why examples are

This seems to hold true for novice programmers as well. In the CPP corpus, the success rate with concrete examples in terms of eliciting ideal answers from students is overwhelming. Across 18 Hailstone dialogues in that corpus, there are 37 uses of concrete examples, not including obligatory opening examples for simulation. Of these, 10 are used for elaboration

1 T And how many different possibilities are there for a game?
2 S **3**
3 T Well, we do have 3 for each player. But there are two players in a game. Let's think about it more thoroughly. How many choices does the user have?
4 S **three**
5 T Very good. And how many choices does the computer have?
6 S **three**
7 T Right. Suppose the user chooses rock. There are different choices available for the computer, so that is 3 combinations.
8 T If the user chooses paper, again, there three choices for the computer. That's 6 combinations.
9 T Lastly, the user can choose scissors. For this, the computer can still pick 3. How many total combinations are there then?
10 S **9**

Figure 4.13: The tutor and student are discussing the number of combinations possible in a game of RPS. Students do not need to think about these possibilities when just playing RPS, but it is necessary to do so to program it (generated by ProPl).

only (e.g., to demonstrate a misunderstood aspect of the problem) and the remaining 27 for elicitation of important observations or abstractions. Of these, the student failed to provide the targeted answer only 4 times (a success rate of 85%).

The four failures in the corpus generally boil down to the abstraction step. Although students in the study were always able to answer the easier lead-in questions, in these failures, they seemed to balk when asked to generalize. In all cases, the tutor gave the answer away, or at least tried to elicit something facile, only to complete it with the more challenging details.

An appealing property of examples for novices is that they allow the tutor to ask isolated and simple questions that the student can answer. Because the tasks in beginning programming assignments are often knowledge-lean, students can usually simulate the desired behavior with little difficulty. Asking the student to self-reflect during the process of a hand simulation can bring important issues to the table, and with the tutor's help, act as a bridge into appropriate planning knowledge. It can also help the student begin to understand that

```
1. preamble/trigger
     a. T asks a question
     b. S answers
     c. T asks follow-up Q, goto (b)
     d. T accepts answer, goto 4(b)
     e. problems with answer, goto 2
2. example/hand calculation subdialogue
     a. T sets the stage
     b. T asks (easy) question
     c. S answers
     d. T evaluates, corrects if necessary
     e. either go to (b) or 3
3. elicit observation
     a. T asks about the example
     b. S answers
     c. T accepts, refines, re-engages 2, or gives answer
4. abstraction/generalization
     a. T connects observation from 3 to code
     b. T asks Q to suggest pseudocode steps or summarize
     c. T evaluates, accepts, refines, or gives answer
```

Figure 4.14: General algorithm the tutor seems to follow when using examples. When necessary (step 1), the tutor engages in an interactive simulation (the example), and "pops" out (step 4) to help the student make the important observation.

the difference between how they solve a problem by hand and how to express that skill as a generalized algorithm. Of course, it is a much greater challenge to express an algorithmic level of understanding. As discussed earlier, examples are used frequently in the corpus – a possible general pattern, including the trigger step for using such an example, is shown in figure 4.14.

4.4.2.5 Paradigm and pseudocode placement The analysis presented in this chapter focuses purely on the times *before* pseudocode is actually added at each stage. A portion of the corpus does involve the tutoring and remediation that involves the layout and organization of the pseudocode. In CPP when a schema is agreed on, the student suggests steps, then places them in the pseudocode. The two most common errors were steps being placed out of order (errors of *arrangement*) and indentation mistakes. Because ProPL was not going to support this kind of tutoring, analysis of these pseudocode placement sub-dialogues was not completed.

It is also important to say more about the programming paradigm. While the focus is clearly on structured programming, the idea of using natural language to elicit program design ideas and decisions would certainly apply to any paradigm, such as object-oriented or functional programming. For example, in their book *How to Design Programs*, Felleisen et. al. present design recipes to help students write functions in Scheme (Felleisen et al. 2001). Each phase in these recipes has the student draw on intuition and use natural language to guide code writing. When students ask for help, most tutorial interaction involves asking general questions about the student's status within the design recipe (Flatt 2002).

4.5 CHAPTER SUMMARY

In this chapter, the pedagogy of Coached Program Planning (CPP) was described: the underlying principles, three-step tutoring algorithm, and elicitation tactics were presented. A summary of CPP is provided below, followed by the main points of the chapter.

CPP is a tutoring model intended to elicit program design ideas using the tacit knowledge of programming to guide the design process. The general aim is to help the student in the transition from understanding the problem to being ready to implement a solution. This is accomplished by asking top-level goal- and schema-eliciting questions and using the student's intuition and commonsense understanding to help them develop a deeper understanding of the problem and how to solve it. A variety of tutoring tactics are used to do this, many of which rely on simulating the desired program behavior, and other forms of example. As a CPP tutoring session progresses, the solution grows as goals are identified and program code fleshed out. This produces a staged solution which simplifies the merging of new program segments into existing code, something novices typically struggle with to a high degree.

CPP is summarized in figure 4.15. The figure shows the top-level tutoring algorithm, elicitation pattern, goals for Hailstone and RPS, and a small selection of tactics described

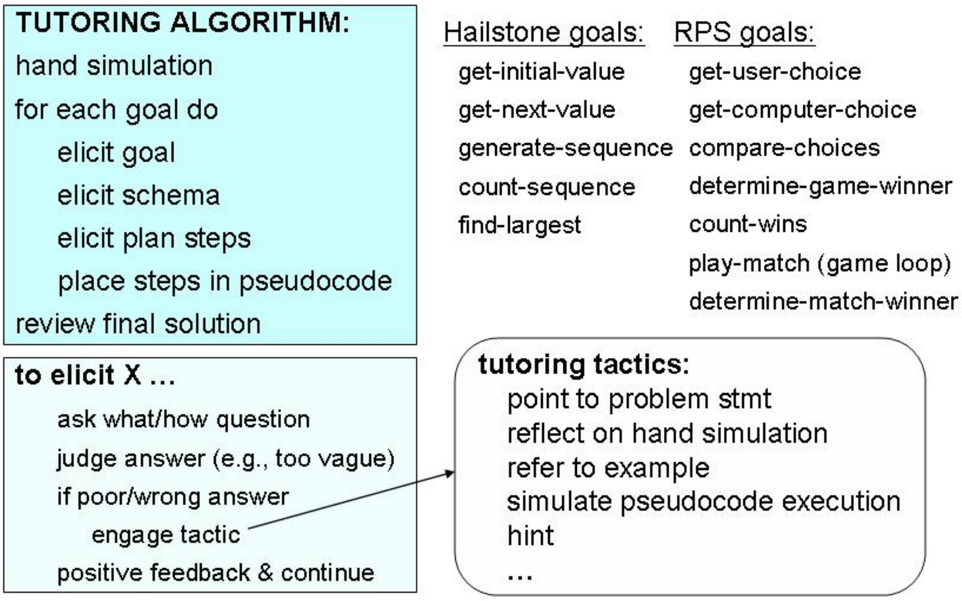


Figure 4.15: CPP in a nutshell.

earlier in the chapter. The main loop of the tutoring algorithm operates over all programming goals of the problem in question. The order of how the goals are achieved depend both on the state of the solution as it evolves and the tutor's discretion (and so this aspect is not addressed in the figure). Similarly, the schemata and plans needed to achieve these goals are not shown because they vary widely and even more options exist with increased independence between the plans.

The main points of the chapter were:

- The problem being addressed by this dissertation is how to scaffold the development of the tacit knowledge of programming through the use of natural language tutoring.

- CPP makes the following pedagogical commitments: adopt a pre-practice model of tutoring, use natural language dialogue as the primary mode of communication, present programs as natural-language-style pseudocode, and teach a staged approach to program design.

- There is a negative transfer from natural language to programming. One aim of CPP is to mitigate this transfer, and help novices learn to understand the difference between languages designed for human communication and those designed for communication with machines.
- CPP targets knowledge-lean problems; that is, problems that require little or no background knowledge to understand.
- Knowledge-lean tasks are often easy to execute by hand, but in many cases, require deeper insight and understanding to express algorithmically.
- A staged approach to programming means that the programmer first writes a program to solve a small sub-problem of the whole, and then repeatedly extends this solution to encompass more and more the whole problem until complete.
- An aim of CPP is to *be present* when novices are first exposed to programming problems and help cultivate the development of their mental libraries of programming techniques through tutoring.
- CPP tutoring tactics are based on the idea of exploiting easy-to-understand properties of knowledge-lean tasks as well as the low-level tendencies of novice programmers to help novices produce the abstractions necessary to write the corresponding algorithm.
- CPP dialogues are governed by a 3-step pattern that prescribes the identification of goals, schemata, and ultimately plans. Each step in the pattern also determines the top-level questions asked by the tutor during a tutoring session.
- The tactics used by CPP fall into two categories: open-ended questions (from the 3-step pattern) and elicitation tactics designed to repair poor utterances or elicit missing concepts.
- Tactics can best be understood by looking at them along three dimensions: purpose, form, and content.
- Using hand calculations is a popular tutoring technique since they allow the tutor to ask simple questions that the student can answer, and ultimately generalize into knowledge suitable for creating a plan.

5.0 ANALYSIS OF STUDENT LANGUAGE

To build the knowledge sources for ProPl, it was necessary to discover how novices respond to the top level questions that ask them to suggest goals, schemata, and plans (from the 3-step pattern). These questions were discussed in section 4.4.1. This chapter provides such a characterization of student answers to these questions from the Hailstone problem dialogues in the CPP corpus.

5.1 HOW STUDENTS DESCRIBE GOALS

To elicit a new programming goal, the tutor typically asks something like "What should we work on now?" Task-wise, these are often *closed-answer* questions, meaning there is only a small number of acceptable answers. However, as shown below, there are many acceptable alternatives for expressing goals. In section 4.2.2, five distinct goals of Hailstone were identified. For each, there is a transition point in the dialogue when the tutor attempts to elicit a new goal from the student. Some examples students' responses to these questions appear in figure 5.1.

Table 5.1 summarizes all of the student answers to such questions from the 16 Hailstone dialogues that were analyzed. For each goal, the table shows typical keywords present in answers, along with their respective frequencies. The first line for each goal also displays the average length in words (right-most column). In some cases, there was no goal-eliciting question-answer adjacency pair, which explains why some of the frequencies are not out of 16 (the total number of dialogues). In such dialogues, the student typically mentioned the goal prematurely, and the tutor used that fact. For example, "Now let's look at that loop you

1. initial-value: “read an integer from a user”
2. next-value: “see whether the number is odd or even”
3. generate-sequence: “check if the number is one that belongs to the ground floor”
4. generate-sequence: “could we do a while loop for odds and evens?”
5. count: “how many numbers there are in the sequence”
6. count: “what to do when the number does equal 4” (outlier)
7. largest: “you need the largest number”
8. largest: “error checking” (outlier)

Figure 5.1: Example goal suggestions from CPP corpus.

Table 5.1: Each section of the table shows commonly used words in students’ answers to goal-eliciting questions.

goal	common answer parts	frequency	avg length
initial-value	*read, prompt, pick, integer, number*	75% (12/16)	6.5
	odd, even, if	19% (3/16)	
next-value	*odd, even, if*	64% (9/14)	11.3
	positive, valid, error	21% (3/14)	
	loop, sequence, ground state	14% (2/14)	
generate-sequence	*loop, repeat, again*	53% (8/15)	13.3
	ground state, 4-2-1, stop	40.0% (6/15)	
count	*count, how many, items in sequence*	88% (14/16)	10.1
largest	*find, largest, highest, maximum*	100% (14/14)	8.0

mentioned earlier.” There are three main observations to make based on the data regarding goals:

Observation #1: Students seem to have little difficulty saying something productive about what goals to pursue.

The high frequency answer categories were all considered valid ways of expressing the goals with which they are associated in the corpus. This is similar to the finding of Spohrer and

Soloway (1985) that novices were generally able to find the right plans for the programs (however they were not able to "put them together"). In other words, it seems that novices are able to determine what needs to be done, both in English and in programs.

Observation #2: If the problem statement contains a clearly stated programming goal, students are better able to state it in the dialogue.

It was pointed out in section 4.4.1.1 that when goals are explicitly stated in the problem statement (such as initial-value, count, and largest), students rarely have a problem stating it. When they do, a simple nudge towards the problem statement is sufficient (this was a tactic discussed in section 4.4).

Observation #3: When goals are not explicitly stated in the problem statement, students resort to either details of the target task or to programming primitives.

Inspecting table 5.1 for the goals not explicitly given (*next-value* and *generate-sequence*), the answer parts for these are not nearly at the level of abstraction of the explicitly stated goals. The tacit goal suggestions (next-value and generate-sequence) tend to be overly specific or programming-related. This behavior suggests that students *talk* about programming in the same way they actually program: bottom-up.

So, what are the implications for building ProPl? To recognize explicit goals, these observations imply that the language of the problem statement needs to be covered. For the tacit goals, the system will will need to have several different elevation tactics at its disposal, and will need to incorporate language involving schemata in order to understand what goals students are attempting to express. This is the topic of the next section.

5.2 HOW STUDENTS DESCRIBE SCHEMAS

Given the nature of students' responses to goal-eliciting questions, it is no surprise that many of those answers lead quickly into a discussion of how to achieve the goal. The typical path the tutor takes in these instances in the corpus is to pursue whatever aspects of the solution the student has suggested. After a goal is established, and if the student has not hinted at

1. generate-sequence: “until a the current sequence number is a 4 2 or 1 keep calculating the next term”
2. generate-sequence: “go back to the if statement” (outlier)
3. count: “use a variable”
4. count: “ok in the loop i’d have a variable starting at 0 that would have a 1 added everytime the loop was executed”
5. largest: “we’ll have to compare each number to the number next to it”
6. largest: “i guess we could see if each number found is bigger than the last and if it is keep it to compare to the next”
7. largest: “we should test it” (outlier)

Figure 5.2: Answers to schema-eliciting questions in the CPP corpus.

an implementation strategy, the tutor generally follows with a schema-eliciting question like “Can you think of a way we can do that?” Although ideal answers on the first try are rare, students again are generally able to “pick out” some aspects of what needs to be said. Some example utterances are shown in figure 5.2. If the student’s answer is flawed, the tutor rarely gives away the answer, opting instead to engage some tutoring tactic to elicit the answer (discussed in section 4.4)

It should be noted that there were no instances in the corpus of the tutor asking a student how to accomplish either the initial-value or next-value goals (other than follow-up elevation tactics – the focus here is top-level responses). This was because the “how” was already evident based on the answers to the goal questions and in the creation of pseudocode. Thus, this analysis consists of the remaining three goals. A summary of all instances of when students were asked schema-eliciting questions appears in table 5.2. Instead of organizing by utterances this time, the table organizes by concepts. This is done because a schema typically consists of several parts. The frequency, therefore, shows how often each concept appears in utterances for that goal. Many utterances contain mention of multiple concepts. For example, the concept of incrementing occurred in 58% of all utterances aimed at describing how to accomplish count-items.

The utterances analyzed in this table represent all initial attempts by a student to talk about “how” to achieve the associated goals. At times, two such utterances were included

Table 5.2: Shows answer parts and their frequencies of student schema suggestions in Hailstone. Boldface entries represent the subsuming concept of the key words. “Other” covers all remaining (rare) concepts.

goal	common answer parts/concepts	frequency
generate-sequence	**repetition**: loop, while, again, go back	100% (19/19)
	termination: ground-state, stop, test	26% (5/19)
	get-next-value: odd, if-statement, next term	21% (4/19)
count	**track**: each time, each step, go through	75% (9/12)
	repetition: loop, while	58% (7/12)
	increment: add 1, increase by 1	58% (7/12)
	save: variable, counter, keep	42% (5/12)
	others	42% (5/12)
find-largest	**compare**: greater-than, less-than, bigger, test	67% (12/18)
	save: variable, remember	50% (9/18)
	track: after each, every value, each, along the way	33% (6/18)
	find-largest (goal): largest, highest, find	33% (6/18)
	initialize: first number, originally set, start out	17% (3/18)
	others	28% (5/18)

(for example, when the tutor pumped). The purpose is to get at the content of how students initially talk about “how.” It should also be noted that in several instances, the student flatly admitted not knowing. This happened a total of six times across these three goals. In a few instances, it was followed by a pump, in which the student *did* turn out to have an idea. These utterances were included in the table.

Table 5.2 confirms a few of the earlier observations made about goals. Although it takes some prodding at times, students still seem generally to have the skill to suggest something relevant and productive. Similarly, because the problem statement does not specifically say anything about *how* to achieve these three goals, student answers are generally incomplete. This is confirmed by noting that many of the frequencies of important answer parts are low. Nonetheless, the data seem to confirm observation #1 that students have the ability to find something productive to say, albeit somewhat harder to elicit in the case of schemata.

Concepts mentioned in “other” categories occurred either once or twice over all utterances. For example, in the count-items group, students mentioned the goal get-next-value, the ground state, finding the largest, and a procedure. While none of these are particularly relevant or required, they are generally ignored by the tutor (no negative feedback). In the case of “procedure,” for example, the student was simply suggesting that a procedure be written to accomplish the counting goal. This is not something the tutor wants to reject – often this is a good way to think about a problem.

What does this imply for schema understanding in ProPl? Most importantly, it means that the system must be aware of all of the various parts of a schema or plan and be prepared to give positive feedback for successful mention of a part. In addition, it must also be able to elicit what is missing. In terms of a traditional dialogue system, a *form-filling* approach to dialogue management (Jurafsky and Martin 2000) is ideal, then, for eliciting schema components from students. In addition to this, another implication is that many of the knowledge structures that are used for schemata will likely be useful for recognizing tacit goal suggestions. This is true since novices tend to jump to the details when asked for such goals.

5.3 CHAPTER SUMMARY

This chapter presented a brief analysis of the language students use in the CPP corpus to talk about goals and schemata. Only data from the Hailstone problem was discussed. The main points of the chapter were:

- Students generally seem to be able to answer open-ended goal- and schema-eliciting questions productively.
- Explicitly stated goals are easy for students to identify, however when asked for tacit problem goals, they almost always resort to low-level aspects of the problem or programming primitives.
- Students are also generally able to identify important aspects of the schemata involved in a solution, but do often omit other important details.

- To understand the “goal-” and “schema-talk” of students, ProPl must be ready to recognize the language used in the problem statement and have a way to recognize individual parts of plans, no matter the order in which they are suggested. This suggests a form-filling approach to handling schema understanding in dialogue.

6.0 AUTOMATED COACHED PROGRAM PLANNING

Obviously, it is not feasible to provide a human tutor for every student prior to every program they write. This is the motivation behind the construction of an automated system capable of performing CPP. Having laid out the underlying pedagogy in the last chapter, this chapter presents ProPl,[1] a partial implementation of CPP. First, ProPl-C, a non-dialogue-based version of ProPl (the 'C' stands for "control") that allows the student to read the tutorial content, is presented. In the discussion of ProPl-C, the focus is on how the interface is designed to convey the pedagogy prescribed by CPP. After this, a discussion of ProPl's tutoring and dialogue capabilities is presented.

As mentioned in the previous chapter, ProPl and ProPl-C are intended to be used by students to help them prepare to write a computer program. The expectation is that students have *not* yet started working on a program prior to using the system. It is best if they have not yet even read the problem statement. The idea is to establish an early tutorial presence and not give students the chance to generate, much less act on, any misconceptions or misunderstandings that may arise.

6.1 PROPL-C: THE CONTROL SYSTEM

The purpose of this dissertation is to explore, define, and evaluate the use of natural language tutoring to teach the tacit knowledge of programming that is known to underly effective program planning skills. To properly perform such an evaluation, ProPl must be compared against a similar system that does *not* use dialogue. In this section, a system to play this

[1] Pronounced "pro-PELL" and short for "PROgram PLanner.

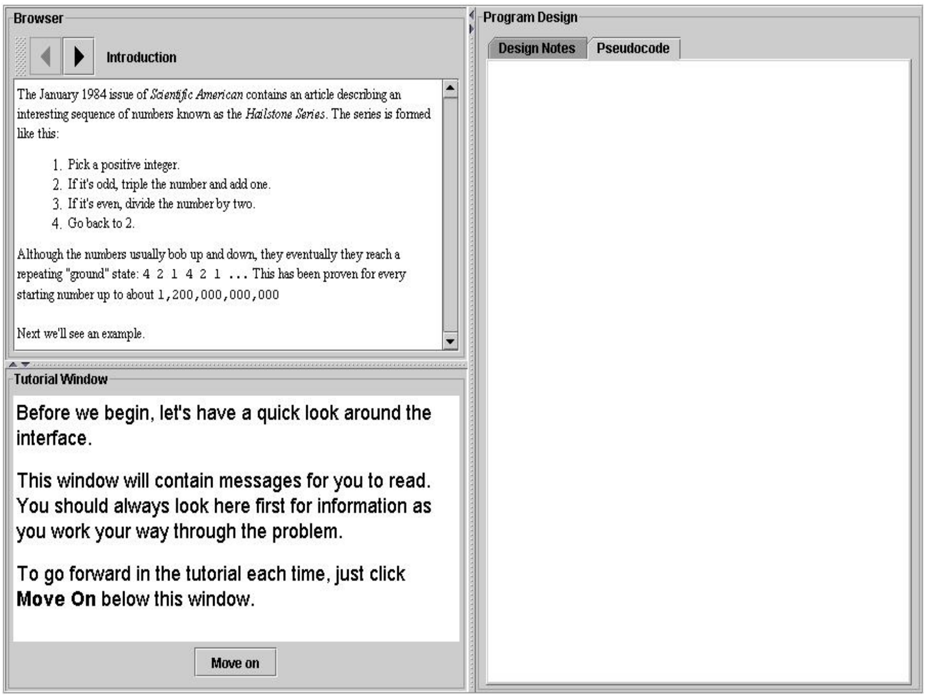
role is shown: ProPl-C. It is used as the control system used in the experiment reported in chapter 7.

To properly attribute any learning benefits to students using ProPl versus those using ProPl-C, differences between the two systems (other than the use of dialogue) must be minimized, hopefully eliminated. As such, there is a great deal of overlap between the two, and so much of the information presented below applies to both systems. Both display the same problem statement, examples, background reading, design notes, and pseudocode. And so, this chapter begins by describing how the pedagogical content of a CPP tutoring session can be delivered in an interface that does not use dialogue.

6.1.1 Interface

ProPl-C runs on any Java-enabled web browser and is connected to a back-end implemented in Lisp that controls the interactions. Using the control system is very simple since it only involves clicking several buttons, tabs, and scrollbars. The interface is very similar to the one used in CPP (see section 4.3.4 and figure 4.4 on page 58) and is shown in figure 6.1. It consists of four primary components:

- **mini-browser** (upper left corner): The problem statement appears here as it normally would appear for the student (say, in a printout). It defines the problem and provides necessary background information. In addition, it is available throughout the session. Back and forward buttons permit viewing of all the pages.
- **tutoring content window** (lower left corner): Tutor messages appear in this window. As the session progresses, the student reads about how to solve the problem, what goals to post, designing schemata, and so on. Once the student clicks “Move On”, this material cannot be reviewed.
- **pseudocode window** (right half): A dual-tabbed pane holding the evolving pseudocode solution.
- **design notes**: The other tab holds important pieces of information discussed in the tutoring content window (more in section 6.1.3).



The January 1984 issue of *Scientific American* contains an article describing an interesting sequence of numbers known as the *Hailstone Series*. The series is formed like this:

1. Pick a positive integer.
2. If it's odd, triple the number and add one.
3. If it's even, divide the number by two.
4. Go back to 2.

Although the numbers usually bob up and down, they eventually they reach a repeating "ground" state: `4 2 1 4 2 1 ...` This has been proven for every starting number up to about `1,200,000,000,000`

Next we'll see an example.

Before we begin, let's have a quick look around the interface.

This window will contain messages for you to read. You should always look here first for information as you work your way through the problem.

To go forward in the tutorial each time, just click **Move On** below this window.

Figure 6.1: The initial screen of ProPl-C, the control version of ProPl.

When using the system, the student is told to read the tutorial content first, follow any instructions it contains (such as reading the problem statement or viewing the pseudocode), and then click the "Move On" button to see the next tutor message. This is the extent of the interaction with ProPl-C.

Sessions with ProPl-C (as well as ProPl) follow the same top-level pattern as prescribed by CPP. In a tutoring session, students move through the following stages:

1. Read the problem statement (which includes an example to read).
2. Do an example (hand calculation) with the system's help.
3. Repeatedly apply the 3-step pattern (goals, schemata, and plans) until the pseudocode is complete.
4. Review the solution as long as desired.

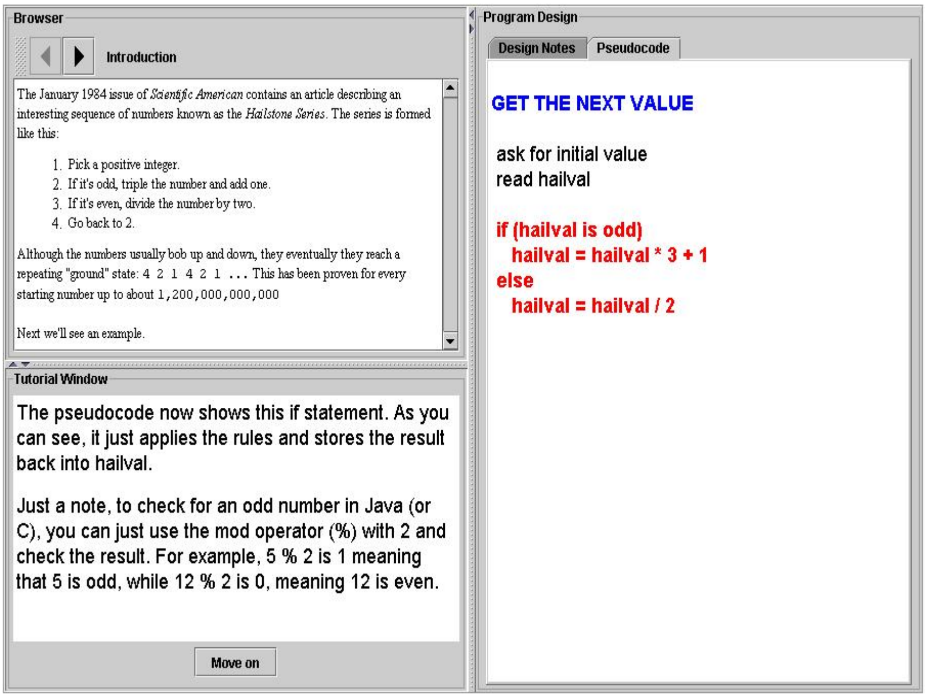
For experimental reasons, students using ProPl-C and ProPl are not allowed to take notes while using the system (this is explained in section 7.2.3.1). At the conclusion, students then are free to write the program as they normally would on their own schedule.

6.1.2 Staged design in the interface

As the student clicks through the tutor messages of ProPl-C, all of the goals, schemata, and plans are described. The messages direct the student through the stages described above, and strictly follow the 3-step pattern, presenting programming goals, followed by schema descriptions, and ultimately the plans for each goal. As shown in figure 6.1, the environment starts with a blank slate. The interface is updated to present the pseudocode incrementally in harmony with the content the student sees. The stages for Hailstone shown in section 4.1.2.4 are the same ones presented by the system.

The first two cases of merging in the Hailstone problem are shown in the two screenshots of figure 6.2 (since the first plan to read input is added to an empty screen, that screenshot is not displayed). The input plan was the first to be posted, which was followed by the conditional statement that obtains the next value. This new code being merged in is shown in a different color to help it temporarily stand out. After a number of "Move On" clicks revealing more tutor messages, the bottom screenshot in the figure shows the state of the program after the generate-sequence code is merged in. At this point, the code satisfying the get-initial-value and get-next-value goals becomes the "old" code, and reverts to the same color (black, in the actual system). The newer steps, that introduce the loop, are merged in and again, show in a bright color. This process continues until the solution is complete.

The pseudocode must obviously be pre-written by a domain author and should be written to use familiar words and phrases to make the connections between it, the notes, and the dialogue as clear as possible. The style of the pseudocode is derived from the solutions developed in the CPP corpus, which in turn was modeled after the pseudocode described in (Robertson 2000).



The January 1984 issue of *Scientific American* contains an article describing an interesting sequence of numbers known as the *Hailstone Series*. The series is formed like this:

1. Pick a positive integer.
2. If it's odd, triple the number and add one.
3. If it's even, divide the number by two.
4. Go back to 2.

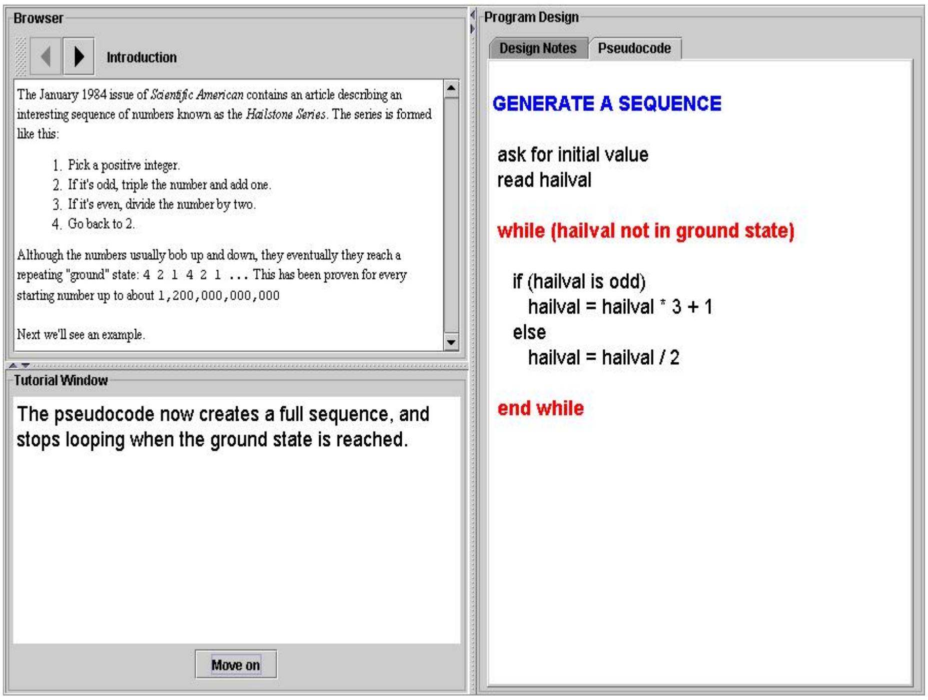
Although the numbers usually bob up and down, they eventually they reach a repeating "ground" state: `4 2 1 4 2 1 ...` This has been proven for every starting number up to about `1,200,000,000,000`

Next we'll see an example.

The pseudocode now shows this if statement. As you can see, it just applies the rules and stores the result back into hailval.

Just a note, to check for an odd number in Java (or C), you can just use the mod operator (%) with 2 and check the result. For example, 5 % 2 is 1 meaning that 5 is odd, while 12 % 2 is 0, meaning 12 is even.

```
GET THE NEXT VALUE

ask for initial value
read hailval

if (hailval is odd)
  hailval = hailval * 3 + 1
else
  hailval = hailval / 2
```



The January 1984 issue of *Scientific American* contains an article describing an interesting sequence of numbers known as the *Hailstone Series*. The series is formed like this:

1. Pick a positive integer.
2. If it's odd, triple the number and add one.
3. If it's even, divide the number by two.
4. Go back to 2.

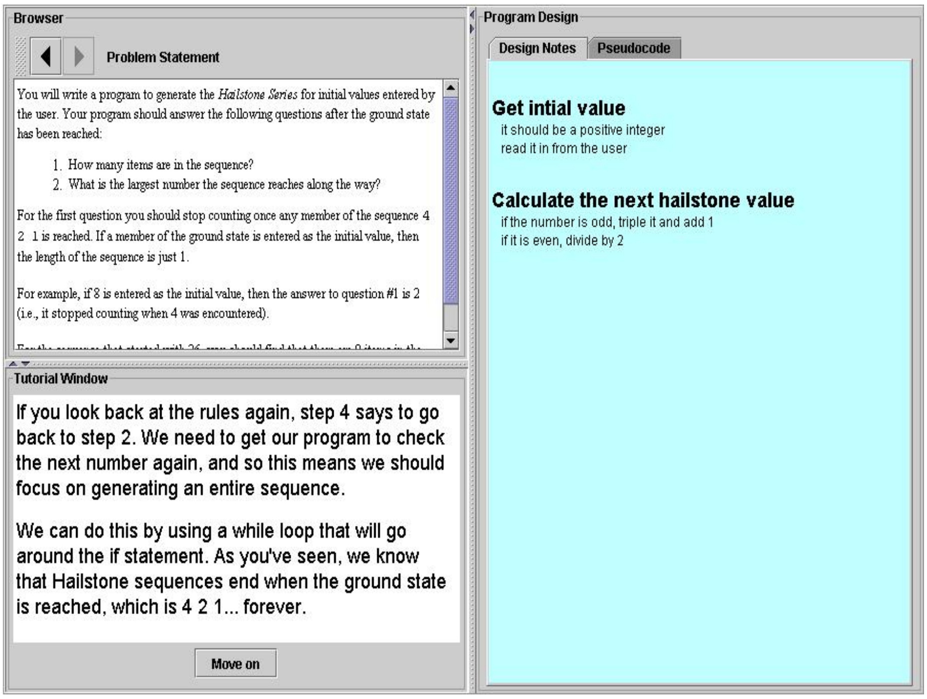
Although the numbers usually bob up and down, they eventually they reach a repeating "ground" state: `4 2 1 4 2 1 ...` This has been proven for every starting number up to about `1,200,000,000,000`

Next we'll see an example.

The pseudocode now creates a full sequence, and stops looping when the ground state is reached.

```
GENERATE A SEQUENCE

ask for initial value
read hailval

while (hailval not in ground state)

  if (hailval is odd)
    hailval = hailval * 3 + 1
  else
    hailval = hailval / 2

end while
```

Figure 6.2: As the student clicks through the tutorial messages, new pseudocode is merged into the existing code each time a new plan is called implemented.



You will write a program to generate the *Hailstone Series* for initial values entered by the user. Your program should answer the following questions after the ground state has been reached:

1. How many items are in the sequence?
2. What is the largest number the sequence reaches along the way?

For the first question you should stop counting once any member of the sequence 4 2 1 is reached. If a member of the ground state is entered as the initial value, then the length of the sequence is just 1.

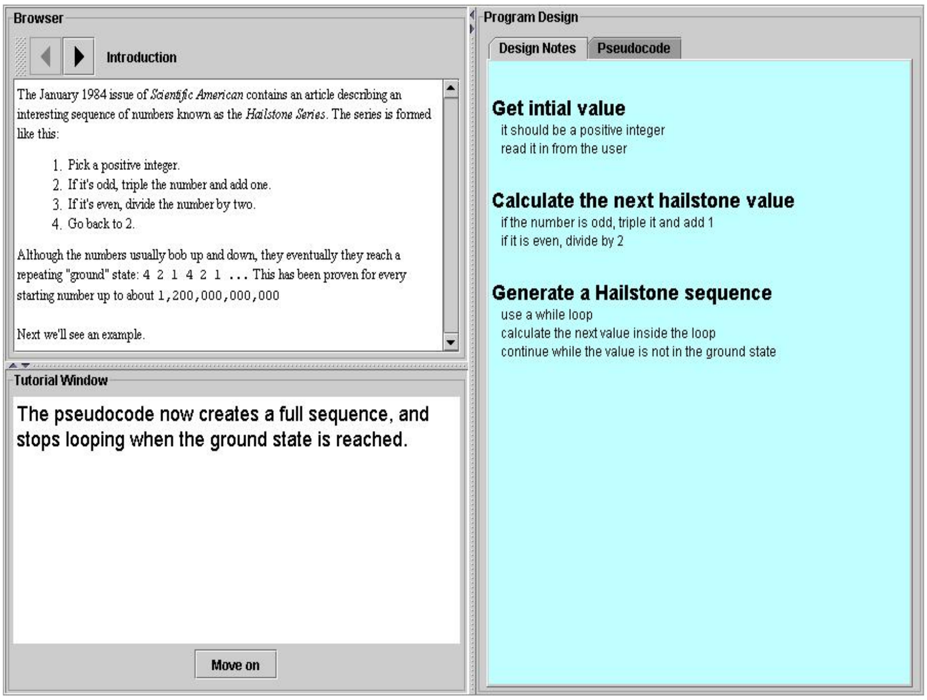
For example, if 8 is entered as the initial value, then the answer to question #1 is 2 (i.e., it stopped counting when 4 was encountered).

If you look back at the rules again, step 4 says to go back to step 2. We need to get our program to check the next number again, and so this means we should focus on generating an entire sequence.

We can do this by using a while loop that will go around the if statement. As you've seen, we know that Hailstone sequences end when the ground state is reached, which is 4 2 1... forever.

Get intial value
it should be a positive integer
read it in from the user

Calculate the next hailstone value
if the number is odd, triple it and add 1
if it is even, divide by 2



The January 1984 issue of *Scientific American* contains an article describing an interesting sequence of numbers known as the *Hailstone Series*. The series is formed like this:

1. Pick a positive integer.
2. If it's odd, triple the number and add one.
3. If it's even, divide the number by two.
4. Go back to 2.

Although the numbers usually bob up and down, they eventually they reach a repeating "ground" state: `4 2 1 4 2 1 ...` This has been proven for every starting number up to about `1,200,000,000,000`

Next we'll see an example.

The pseudocode now creates a full sequence, and stops looping when the ground state is reached.

Get intial value
it should be a positive integer
read it in from the user

Calculate the next hailstone value
if the number is odd, triple it and add 1
if it is even, divide by 2

Generate a Hailstone sequence
use a while loop
calculate the next value inside the loop
continue while the value is not in the ground state

Figure 6.3: Design notes, which show program goals and paraphrases of the schemata and plans involved in achieving them, are shown for the next-value and generate-sequence goals of the Hailstone problem.

6.1.3 Design notes

As the student progresses through the messages, the other tab on the right half of the interface also fills with *design notes*, which are essentially a record of important observations made in the tutor messages. They consist of:

- The goals of the problem, as staged programs to write (represented by bold face, larger lines in the interface).
- Comments about the schemata and plans needed to achieve those goals (represented by pre-authored English sentences similar to the tutorial content).

The comments appear below each goal they elaborate. The goals and comments are linked to certain tutorial messages and appear immediately after the student has read about them. When authoring the design notes, domain authors should include all of the programming goals, the need for particular programming constructs, and any other useful observations that can be tied to the dialogue content. Figure 6.3 contains the corresponding screenshots from figure 6.2, showing the notes after the `next-value` and `generate-sequence` goals are satisfied.

Although there is no practical reason to post such notes, they are included it to act as a representation and reminder of the usually tacit knowledge that goes into producing the final solution. The *reification* (i.e., physical posting) of problem solving actions or decisions that otherwise remain hidden or implied is common in many intelligent tutoring systems (e.g., Singley 1990). As mentioned earlier, the benefit of using natural language to reify the tacit knowledge of programming is that the student can understand without having to learn a new intermediate representation or artificial planning language.

The student is free to move between viewing the pseudocode or design notes at any time by clicking on the corresponding tabs. The system will, however, change the screen back to the pseudocode screen and pause before displaying new pseudocode to help the student see the integration of a new plan. In CPP, full pseudocode development was supported, with full dragging and dropping of steps. Because of the computational and reasoning demands this posed for ProPl, it was dropped in favor of the approach described above. This does, however, remain as future work, and is discussed in chapter 8.

6.2 PROPL: A DIALOGUE-BASED ITS FOR NOVICE PROGRAM DESIGN

ProPl-C was, in fact, built after ProPl. Much of the same source code is used, minus the dialogue-enabling components. They both teach the same pedagogical content, including everything discussed in the previous section. The key difference between the two is that ProPl attempts to *elicit the design elements via dialogue* rather than provide them in reading (i.e., "mini-lessons").

6.2.1 What happens in a tutoring session

A student using ProPl follows the same top-level pattern as listed above (problem statement, hand calculation, and so on). Interaction is vastly different, however, since after reading the problem statement, the system begins asking questions. The first dialogue the student has with the system is to do an interactive hand calculation. An example of such an interaction with ProPl appears in figure 6.4 (the dialogue is from the ProPl corpus). The primary purpose of the hand calculation is to confirm the student's understanding of the task targeted by the problem statement. As shown in the figure, the dialogue is usually very simple, asking the user to simulate the input-output behavior of the desired program. During this interaction, the student is not asked about programming goals or schemata, just the task at hand. In the RPS dialogues, the student is presented with game choices and asked to report the outcome of games. It is generally very easy to conduct such dialogues for knowledge-lean tasks such as these.

Although errors are rare in the hand calculations, minor slips do occur. The student in this example makes two mistakes, both of which are corrected by ProPl. On line 6, the student incorrectly applies the rules to the value of 3. A similar error happens on line 10. In both cases, the student seems to recover by providing the correct next answer. This student was able to provide the correct count and largest value for this sequence in the ensuing dialogue. It should also be noted that had the student said the ground state was reached in lines 16, 18, or 20, ProPl would have stopped the example and confirmed that

1 T If we start at 12, what comes next?

2 S **6**

3 T Fine. What comes after 6?

4 S **3**

5 T Good. Ok, what follows 3?

6 S **5**

7 T 3 times 3 is 9, then you add 1 to get 10. after 10?

8 S **5**

9 T Super. good. next one after 5?

10 S **8**

11 T Well... Since 5 is odd, we triple that and add 1 giving 16. after that?

12 S **8**

13 T Good job. yep. what's next?

14 S **4**

15 T Yep. after 4?

16 S **2**

17 T ok, after 2?

18 S **1**

19 T after 1?

20 S **4**

21 T Do we need to keep going?

22 S **no**

23 T That's great. The sequence will now repeat 4 2 1 (the ground state) forever, so we can stop.

Figure 6.4: ProPl performs an interactive hand calculation of a Hailstone series with the student.

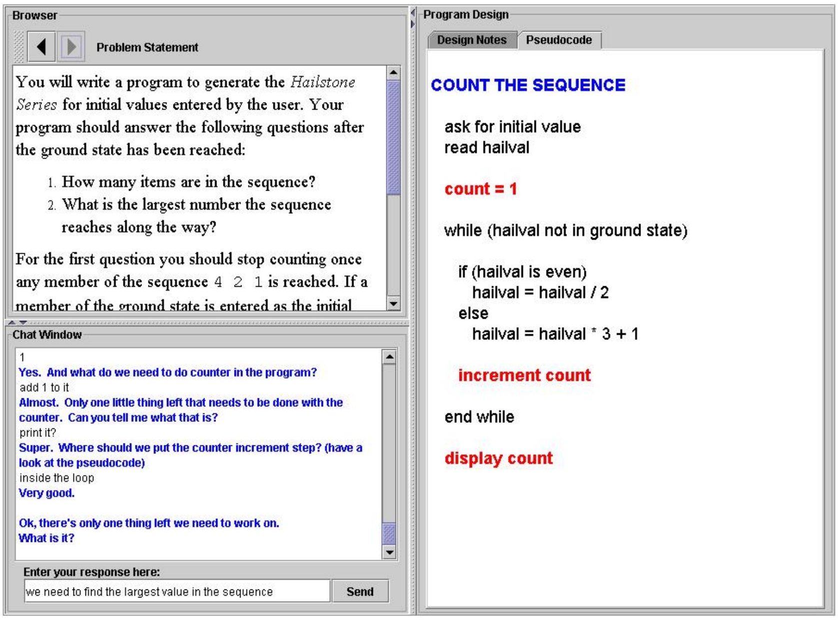
it was ok to stop. The major benefit of going through such an exercise is that students who tend to be careless get a chance to make slips and mistakes with the tutor present. Serious misunderstandings of the problem and task have a far less chance of carrying through to implementation because of this.

Once the simulation is completed, ProPl begins by asking top-level questions as prescribed by the CPP 3-step pattern. As each programming goal is correctly identified during the course of a tutoring session, it is posted in the design notes pane. After a goal is identified, a "how" question follows and, when answered correctly, aspects of the schemata that achieve these goals are posted as comments beneath the relevant goals. To complete the 3-step pattern, dialogue ensues to describe the needed plan, and the pseudocode screen is updated. It is important to note that the student plays no role in the actual construction of the pseudocode, other than being asked where steps should go. Although it would be highly desirous to have the student deeply involved in this part of the planning, it was soon discovered to be too difficult a problem to deal with in a general way, and to support it in dialogue.

As the problem solving session evolves, both the design notes screen and pseudocode screen grow, just as it does in ProPl-C. When students are unable to answer a question, or give a wrong answer, ProPl continues by engaging in a subdialogue aimed at eliciting the desired answer. These tactics were described in section 4.4. When the student is unable to provide the correct answer even after the remedial tutoring tactics, ProPl will eventually give it away. The student is told when the last goal has been achieved in the pseudocode and given a chance to review the solution and notes as long as desired. In addition, the browser and dialogue history are also reviewable at any time.

The interface for ProPl is shown in figure 6.5. As the figure shows, it is nearly identical to that of ProPl-C. The only difference is in the lower left hand window: the tutorial content window is replaced with a dialogue window. Tutor utterances appear in boldface and the student responds by typing into the response field. This is identical to how interaction occurred with the human tutor in the CPP environment.

Although there is a period of adjustment and learning with any new system, this time is especially lengthy for dialogue systems (interacting with the system, how to phrase responses, understand the system's output, etc.). As such, before using ProPl for for the first time, users receive a short tutorial for a simple three line program that computes a person's body mass index. The tutorial demonstrates how to answer questions and work in the environment. Participation is limited to clicking buttons and typing predetermined answers to the questions. Providing such a tutorial is known to be an effective way to accelerate users' abilities to use dialogue systems effectively (Kamm, Litman, and Walker 1998).



You will write a program to generate the *Hailstone Series* for initial values entered by the user. Your program should answer the following questions after the ground state has been reached:

1. How many items are in the sequence?
2. What is the largest number the sequence reaches along the way?

For the first question you should stop counting once any member of the sequence `4 2 1` is reached. If a member of the ground state is entered as the initial

1
Yes. And what do we need to do counter in the program?
add 1 to it
Almost. Only one little thing left that needs to be done with the counter. Can you tell me what that is?
print it?
Super. Where should we put the counter increment step? (have a look at the pseudocode)
inside the loop
Very good.

Ok, there's only one thing left we need to work on.
What is it?

we need to find the largest value in the sequence

```
COUNT THE SEQUENCE

ask for initial value
read hailval

count = 1

while (hailval not in ground state)

  if (hailval is even)
    hailval = hailval / 2
  else
    hailval = hailval * 3 + 1

  increment count

end while

display count
```

Figure 6.5: PROPL's interface uses a dialogue window (lower left corner) to interact with the student. Tutor utterances appear in blue and the student's in black.

6.3 DIALOGUE ENGINE AND KNOWLEDGE SOURCES

PROPL is an application of the Atlas dialogue management system (Freedman et al. 2000), a domain-independent framework for the development of natural language dialogue systems. Atlas provides two main components:

- **Atlas Planning Engine:** (APE) a planner for tutorial dialogue.
- **CARMEL:** a natural language understanding component which includes robust and efficient algorithms for parsing, semantic interpretation, and repair.

To build a tutoring system using Atlas, it is necessary, at a minimum, to provide a plan library to guide tutorial interactions and a semantic grammar and lexicon for the sentence-level understander. Atlas also prescribes the use of *Knowledge Construction Dialogues* (KCDs) when developing and provides additional tools to support their creation and refinement (Rose et al. 2001; Jordan, Rose, and VanLehn 2001).

6.3.1 Knowledge Construction Dialogues (KCDs)

Briefly, a KCD is based on a main line of reasoning that it elicits from the student in a series of questions. If understanding fails to detect a correct answer to a question, it initiates a subdialogue, which can be another KCD or a bottom-out utterance giving away the answer. Different wrong answers can elicit different subdialogues to remedy them, and there is always a generic remedial subdialogue for answers that cannot be recognized as correct or as one of the expected wrong answers. The dialogue management approach for KCDs can be loosely categorized as a finite state model (Jurafsky and Martin 2000). Tutor responses are specified in a hand-authored network. State nodes in the network indicate either the system should question the student or push and pop to other networks. The links exiting a state node correspond to anticipated student responses to the question. Anticipated student responses are recognized by looking for certain phrases and their semantic equivalents (Rose et al. 2001).

ProPl contains about 35 top-level KCDs and 100 more remedial KCDs covering three programming problems (see section 7.2.3.1). As mentioned earlier, when moving through the 3-step pattern, each step engenders a question to the student. These are the first questions asked in each top-level KCD (minus those doing the initial hand simulations) and the student's answers dictate which remedial subdialogues are called, if any. There are KCDs for all goals and plans involved in each solution. KCDs to elicit goals are often short while those to elicit plans are longer because each part of the plan must be addressed.

KCDs in ProPl were constructed with a variety of goals in mind. For example, many involve the refinement of vague answers, completion of incomplete answers, and redirection to concepts of greater relevance. ProPl's responses are dictated entirely by the authored content of the KCDs. That is, the system's responses to student answers are dictated by the classification of that answer by the CARMEL and the KCD "called" for that answer class. There is no higher level tutorial decision making or student modeling done in ProPl, even though APE provides some functionality for introducing advanced techniques such as this into the planning operators. A sample of KCDs for the Hailstone problem are shown in appendix D.

6.3.1.1 Effort required for a new problem KCD authoring task is much simpler with a corpus since expected answers and the problem-specific aspects of tutoring are readily available. It is still certainly possible to write KCDs for a new problem without it, however. The Snap KCDs were written with no corpus to draw from and in the KCD analysis presented in the next chapter (section 7.4), this set of KCDs did just as well as questions from the Hailstone and RPS problems. On the other hand, at the point the Snap KCDs were written, I had a great deal of experience having gone through the full process twice. It made a huge difference to have had access to example questions and answers when working on them.

The general KCD engineering process consisted of first gathering a collection of student answers to the goal- and schema-eliciting questions (when a corpus was available). The tutoring tactics were then identified (either through creativity or the corpus) and answer classes identified, then implemented in KCDs. In my experience, the rough estimate of time needed to author each top-level KCD, along with the shorter ones that "hang" off them, was about 2.5 hours. The "large" KCDs implementing advanced tactics took between a 30 and 120 minutes each (there were a few of these in each of the three problems). These are estimates based on my reflection on the process. Also, the time is greatly dependent on the desired level of breadth of understanding and tutoring. Total effort, including building up the NLU sources, integrating them in with the ProPl environment, revising them after test runs, and writing the necessary HTML content, it was about an average of 60 hours per problem (overall total of 180). Of course, the key benefit is that once a problem is authored, it is available forever (and can be iteratively improved over time).

6.3.2 Top-level dialogue control

Top-level control of ProPl consists of a set of simple APE operators that call KCDs in a pre-determined order. For each problem, KCDs exist to elicit each goal and schema/plan. These are called in the same order they are presented in ProPl-C, which is also the same order students seemed to prefer in the human-human corpus. The completion of a KCD triggers the update of the design notes and pseudocode windows, depending on if a goal or schema was just completed.

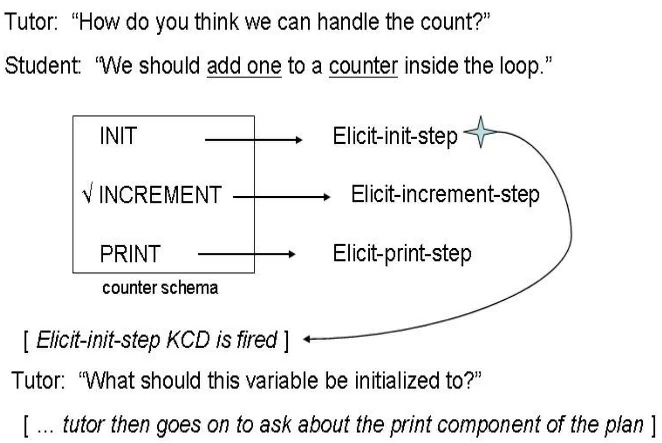

Tutor: "How do you think we can handle the count?"
Student: "We should add one to a counter inside the loop."
INIT
√ INCREMENT
PRINT
counter schema
Elicit-init-step
Elicit-increment-step
Elicit-print-step
[Elicit-init-step KCD is fired]
Tutor: "What should this variable be initialized to?"
[... tutor then goes on to ask about the print component of the plan]

Figure 6.6: ProPl handles many "how" questions by checking for parts in the student's answer, then calling remediation KCDs for the missing elements.

6.3.3 Understanding student input

The goal/schema distinction in CPP provides a useful context for understanding. For example, if the tutor asks a goal-eliciting question, the understanding task is to infer the relevant goal from the student's answer. The aim is to take every advantage of the "goodness" present in utterances. Similarly, when students answer schema-eliciting questions, they seem to "pluck" concepts from various sources, like the problem statement or their limited (largely syntactic) understanding of programming. Nonetheless, in the human-human dialogues, novices are generally able to say *something* correct, even if it sometimes was only "in the ballpark" (see chapter 5).

It is therefore quite important to identify the correct components in an answer and provide feedback indicative of that recognition. It is important not only to carry on natural-sounding dialogues, but also to maintain student morale (Lepper et al. 1993). Two strategies are used in ProPl to handle recognizing a wide-range of correct answers:

1. Build up the semantic grammar with utterances collected in the human-human corpus, pilot testing, and from subject-matter experts (a standard practice).

2. Use the *answer part* facility provided by the KCD component of Atlas to allow the student to mention schema components in any order (depicted in figure 6.6).

Building up a semantic grammar – that is, expanding the breadth of key words and phrases that can be recognized – is necessary when building most dialogue systems. Luckily, Atlas provides several tools to help automate the process. The second method, using answer parts to implement form-filling, represents an unintended application of the method in ProPl as it has been used in other KCD-driven systems (Rose et al. 2001). Answer parts are typically authored when a student's answer is expected to contain mention of all parts *on the first try*. In ProPl, this assumption is relaxed by turning off negative feedback when parts are left out. Instead, KCDs are called that elicit fillers for the "gaps" in the student's original answer. In dialogue systems research, this strategy is known as a *form-filling* approach to dialogue management. In sum, it permits students to talk about any component of a schema/plan, with other KCDs being called to elicit the remaining details.

6.3.4 Differences between ProPl-C and ProPl

The key and only intended difference between ProPl and ProPl-C is the mode of communication with the student. More specifically, ProPl uses free form natural language dialogue for communication and ProPl-C uses reading alone. As mentioned, the problem and solution content is identical. Nonetheless, there are some unintended differences between the two systems that are worthy of mention. These are particularly important since it is these two systems which are pitted against each other in the evaluation described in the next chapter.

The tutor messages in ProPl-C are intended to replace the dialogue window of ProPl. The pedagogical content of the authored messages was written to represent faithfully what the dialogue covers, but it would be impossible to do so perfectly. It would be overwhelming and likely not comprehensible to show the possible remedial KCDs that exist in ProPl. Another strategy is to perform *yoking*, which simply means that dialogues from the experimental group are shown as reading for the control group. Although this is a good solution, the decision was made to adopt a more palatable, straightforward approach of canned reading (or "mini-lessons"). This approach has been used in previous studies on KCDs (Rose et al. 2003a).

Another difference arises when one considers the time it takes to use each system. Since a ProPl-C student is only reading and is not required to type answers from scratch, it naturally takes them less time to make it through a complete solution. Time-on-task differences are common in studies of educational systems, and so the decision was made to not introduce some artificial time filler or restrictions on using either system. In fact, when one looks at the overall times, including time on the compiler and planning time, it is not clear at all there is even a time-on-task issue with these systems.

6.4 CHAPTER SUMMARY

This chapter discussed automating CPP. The main points of the chapter were:

- ProPl is a partial implementation of CPP. Goals, schemata, and plans are elicited, but very little pseudocode arrangement is performed. Rather, it is presented to the student stage by stage, in tandem with the progress of the dialogue.
- ProPl-C is a non-dialogue version of the system that presents a rendition of the dialogue content as mini-lessons. The same solutions are presented by both systems.
- Design notes are English summaries of the goals and schemata involved in a solution. Their purpose is to reify the tacit knowledge of programming in the context of the problem being solved with the system.
- ProPl tutoring sessions follow the form prescribed by CPP: an initial hand calculation is done, followed by progression to the 3-step pattern to produce the pseudocode.
- ProPl is an application of Atlas, a dialogue toolkit for tutoring systems. The primary knowledge source for ProPl are *Knowledge Construction Dialogues*, which are hierarchical structures that walk student through directed lines of reasoning.
- ProPl handles understanding by possessing a large semantic grammar for programming and using the answer-part facility of Atlas. This allows the KCDs to handle a variety of student answers, and use a form-filling approach to handle schemata.

7.0 EVALUATION

In this chapter, two controlled evaluations are reported: one comparing students receiving CPP from a human tutor versus those receiving no tutoring at all, and another comparing ProPl and ProPl-C. The first experiment took place in fall of 2001 and spring of 2002 and used introductory programming students as subjects. The study was designed with two primary goals in mind:

1. Collect a corpus of human tutoring sessions to inform the development of an intelligent tutoring system for beginning programmers.
2. Evaluate the impact of CPP on students' programming ability.

The corpus analysis focused on analyzing the tutoring tactics as well as the kinds of answers and general language students use to talk about programming. Section 4.4 described the tutoring tactics used in the corpus and in chapter 5, a brief overview of how novices responded to goal- and schema-eliciting questions was presented. Regarding the evaluation of CPP on students' programming ability, this study was very much an exercise in determining how best to go about it given the aims of CPP.

The second experiment, which took place in the spring of 2004, refines and improves the first. It pits ProPl against ProPl-C with students coming from three introductory programming courses. Again, projects from the courses were used as subject matter, but this time written pre- and posttests were given to improve the evaluation. The most important finding from this experiment, and perhaps the entire dissertation, is that ProPl students displayed more skill at solving the composition problem. Sadly, there is a certain amount of discord between the two experiments. Certainly mistakes were made in the first experiment, but part of this effort was exploratory: it was not clear, in the beginning, what (if any) impact pre-practice tutoring would have on students. Luckily, the results of the CPP study led to revisions and refinements that made the ProPl study more coherent and targeted.

7.1 CPP VERSUS BASELINE

A pilot study was conducted in the summer of 2001 to test the CPP software and begin collection of the corpus. Tutoring occurred with a human tutor[1] over a network. Student reaction was very positive. Based solely on their reactions, the decision to move forward towards ProPl was essentially assured.[2] Nonetheless, to build up the corpus and to begin assessing what impact it might have on beginning programmers, the controlled evaluation moved forward.

7.1.1 Design

Students began by signing consent forms and reading about the experiment. Next, they were assigned to one of two conditions according to their preferences: the experimental condition with subjects using the CPP environment over a network with a human tutor or a control condition allowing students to go about writing their programs as they normally would (with no intervention). It was, perhaps, a mistake to not assign students at random, but given the goal of building up a corpus, those students willing to spend the extra time to come in for tutoring (and get paid, not all were willing) were not turned away.

7.1.2 Participants

Subjects were volunteers from *Introduction to Computer Programming*, a CS0[3] service course at the University of Pittsburgh and were paid $7/hour for their participation. 16 students

[1] The author was the tutor.

[2] Immediately after the tutoring session, one student sat back from the computer and said, "Cool! I've already started on my program!" The comment was not simply enthusiastic, but suggested the student perceived pseudocode planning *as programming*.

[3] This is the general ACM classification for beginning programming courses that are not intended for computer science majors (which is labeled CS1).

volunteered (out of roughly 35 in the course) and 2 were removed for not completing the course. None of the subjects had programming experience beyond a few weeks of BASIC or spreadsheet skills. The initial assignment of students was $n = 7$ to each condition.

7.1.3 Materials

Four projects from the course were used in this study, starting roughly one month into the course.

1. **Numbers to Words**: input a number between 1 and 999 then display the number in English words.
2. **Hailstone**: shown in figure 4.1 (page 47).
3. **Rock-Paper-Scissors** (RPS): play a "best-of" match of RPS games, keeping track of wins and picking randomly for the computer. The solution involved the use of subprograms (appendix A).
4. **Lotto Check**: an array program requiring a variety of array processing activities.

The first assignment acted as the pretest for students in both groups; no tutorial intervention was performed. Students in the CPP group received tutoring for the Hailstone and RPS problems before they attempted to implement solutions. Finally, no intervention was given on the fourth project allowing it to play the role of posttest.

7.1.4 Procedure

The experiment began roughly one month into the semester. In their classes, students had learned the basic concepts of computer systems (memory, processors, etc.) and some rudimentary programming concepts including types, variables, operators, and simple input/output. The study spanned nearly six weeks, during which students learned control flow (conditionals and loops) and how to write simple subprograms. All students signed a consent form agreeing to participate in the study and have data about them collected and analyzed. Data collection consisted of recording compiler and editor activities of all subjects. This data also included copies of all files submitted to the compiler – this collection of files is called an *online protocol* (this method was also used in Spohrer and Soloway 1985). In addition, students were allowed to take notes as well as a copy of the pseudocode after their tutoring sessions.

7.1.4.1 Floundering To determine if CPP subjects displayed less erratic behavior during implementation, the idea of *floundering* was borrowed from the fields of user interfaces and intelligent tutoring systems. Students are deemed to flounder when their movements in an interface begin to seem random and often rapid. A student with no idea how to move forward will either stop altogether or resort to floundering. In programming, Perkins (1989) refers to these two groups as *movers* and *stoppers*. An extreme mover, a *tinkerer*, is one who thinks very little about the purpose behind each compile attempt, and usually hopes (rather than knows) if each change will bring the desired results.

Floundering is defined in programming as repeated attempts to repair an algorithm-related bug (as opposed to a syntactic or language-construct bug) that leaves the program no closer to being correct after each attempt. To gauge floundering, the online protocols were tagged based on the content of the differences between each pair of syntactically correct versions of the programs. Changes between each pair of files were labeled as involving an *algorithmic bug-fix* or something else.[4] Basically, if a compile attempt appears to be grounded in an algorithmic problem, and it did not show signs of progress towards correction of the error, then it was tagged.

Under this definition, tinkerers will produce more instances of floundering than students closer to the stopper end of the spectrum. So, instead of counting the raw number of compiles that are classified as floundering, the solution was to count the number of floundering *episodes*. An episode is defined as one or more consecutive recompiles intended to fix the same error. This levels the playing field between the different classes of students. For example, an episode of length 17 and one of length 3 will each count as one episode. Intuitively, then, the number of floundering episodes found in an implementation represents the number of algorithmic impasses the student struggles to repair over the course of a protocol.

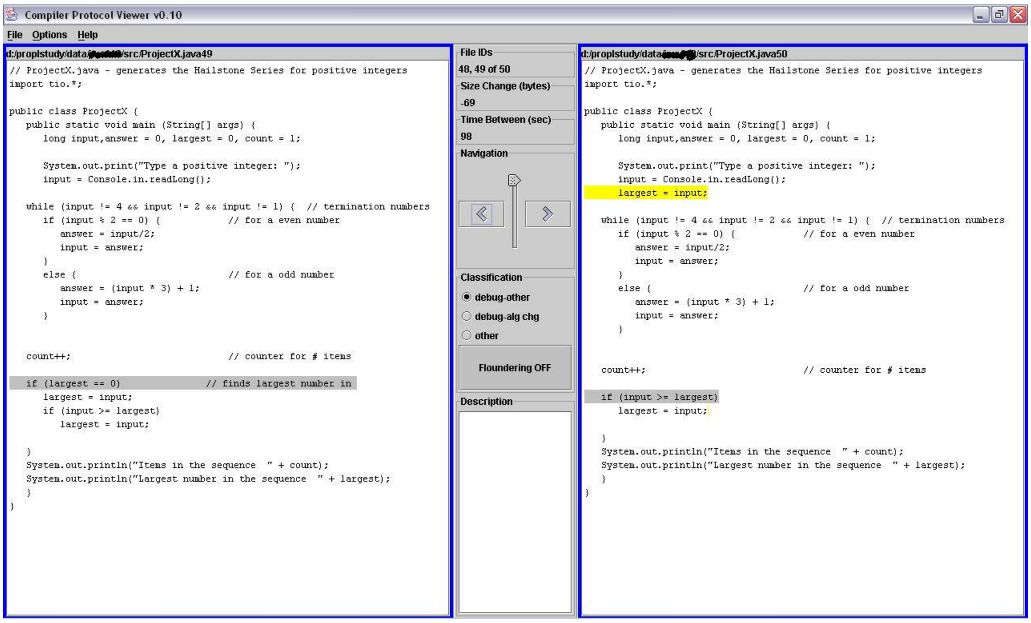
[4] “Something else” includes things like tweaking i/o behavior, adding comments, superficial rearrangement of program code, and adding large code segments.

Two other measures were used to evaluate CPP's impact. To see if students who used CPP were more included to use comments, the number of comment lines in the final version of the posttest program was counted. Single line comments counted as 1, as did each line within multi-line comments. Administrative comments (name, date, class, etc.) were not counted. This metric was largely motivated by my own observations that students did not use comments to sufficient levels, and was confirmed by a number of other instructors. My belief was that the use of natural language dialogue to prepare for a program could, perhaps, carry over into an implementation in the form of more use of comments. Second, to determine if CPP subjects gained a better understanding of structural concepts, all files submitted to the compiler, including those with syntax errors, were checked for indentation errors. Each improperly indented line was counted upon its first appearance. The rules for proper indentation were liberal (one space was considered enough), but consistent with those presented in popular introductory programming textbooks.

7.1.5 Results

Because of a possible self-selection bias (subjects were assigned to one of the two conditions according to their stated preferences), the first project included in the study, also the first non-trivial assignment given in the class, was used as a pretest. The scores on these projects (from the course) revealed no significant association between condition and pretest ($t(12) = -0.269$, $p = 0.79$), allowing the conclusion that both conditions consisted of subjects of equivalent competence. In analyses below, then, ANCOVAs are used to factor out the pretests and support the conclusion that the intervention was responsible for the differences and not student ability.

Floundering The first test for floundering was to simply count all successful compile attempts on the fourth assignment in the study (covering arrays, untutored). CPP subjects compiled an average of 25.0 ($SD = 32.9$) times on this project, and control subjects compiled 49.4 ($SD = 41.38$) times. The difference is not statistically significant. However, by throwing out a statistical outlier in the CPP condition who was 2.2 standard deviations from the mean, the difference becomes marginally significant ($F(1, 10) = 4.08$, $p = 0.071$). The



d:/proplstudy/data/[redacted]/src/ProjectX.java49

```
// ProjectX.java - generates the Hailstone Series for positive integers
import tio.*;

public class ProjectX {
   public static void main (String[] args) {
      long input,answer = 0, largest = 0, count = 1;

      System.out.print("Type a positive integer: ");
      input = Console.in.readLong();

   while (input != 4 && input != 2 && input != 1) {  // termination numbers
      if (input % 2 == 0) {            // for a even number
         answer = input/2;
         input = answer;
      }
      else {                           // for a odd number
         answer = (input * 3) + 1;
         input = answer;
      }


   count++;                            // counter for # items

   if (largest == 0)               // finds largest number in
      largest = input;
      if (input >= largest)
         largest = input;

   }
   System.out.println("Items in the sequence  " + count);
   System.out.println("Largest number in the sequence  " + largest);
   }
}
```

File IDs: 48, 49 of 50

Size Change (bytes): -69

Time Between (sec): 98

Classification:
● debug-other
○ debug-alg chg
○ other

d:/proplstudy/data/[redacted]/src/ProjectX.java50

```
// ProjectX.java - generates the Hailstone Series for positive integers
import tio.*;

public class ProjectX {
   public static void main (String[] args) {
      long input,answer = 0, largest = 0, count = 1;

      System.out.print("Type a positive integer: ");
      input = Console.in.readLong();
      largest = input;

   while (input != 4 && input != 2 && input != 1) {  // termination numbers
      if (input % 2 == 0) {            // for a even number
         answer = input/2;
         input = answer;
      }
      else {                           // for a odd number
         answer = (input * 3) + 1;
         input = answer;
      }


   count++;                            // counter for # items

   if (input >= largest)
      largest = input;

   }
   System.out.println("Items in the sequence  " + count);
   System.out.println("Largest number in the sequence  " + largest);
   }
}
```

Figure 7.1: Tagging tool for online protocols.

second test was to count episodes of algorithmic floundering. In the posttest assignment, CPP subjects had a mean of 1.00 ($SD = 2.24$) episodes and the control subjects averaged 2.71 ($SD = 2.63$). The difference is marginally statistically significant ($F(1, 11) = 3.53$, $p = 0.087$) and moderately large (effect size = 0.65).[5]

Because floundering is a subjective measure, the online protocols were coded independently by two experienced programming instructors. A coding tool was developed to help the coders tag the protocols (a screenshot appears in figure 7.1). It displays programs in pairs and provides button- and slider-based navigation for easy viewing. In the center of the interface, coders select the appropriate tag for each pair of programs (the kind of debugging act and whether or not it can be classified as floundering). In addition, differences between the two versions of the program are highlighted (with the help of `diff`) to help the coders see where changes occur.

[5] Effect size was computed using Glass' delta, that is, $\frac{M_{exp} - M_{ctrl}}{SD_{ctrl}}$

To develop the conditions and understanding of floundering, four randomly chosen protocols were used for training and discussion. Coding was done together on these four leaving the remaining protocols to be coded independently by each coder. After identifying the beginning and ends of all floundering episodes, the inter-coder reliability was computed. To do this, the *kappa statistic*, a measure popular in the computational linguistics community was used (Carletta 1996). Using a $+/-1$ cushion on the boundaries of episodes 2 in length or longer, the resulting kappa value was 0.872.[6] This means the identification of floundering can be considered a consistent and usable measure to gauge a student's success in developing a program.

Commenting In their final posttest programs, CPP subjects had a mean of 35.0 ($SD = 23.1$) comment lines. The mean for control subjects was 12.3 ($SD = 9.76$), leaving a difference of nearly 23 more lines of comment lines on average for CPP subjects over control subjects. This difference is statistically significant ($F(1, 11) = 5.97$, $p = 0.0325$) and large (effect size = 2.33).

Indentation Because indentation had been covered in class and gradually learned throughout the semester, there was a ceiling effect when analyzing the indentation of the posttest program. Thus, results for the second project are presented understanding that the CPP students had already created a pseudocode solution prior to their real solution. During the implementation of the second project, CPP subjects produced an average of 2.43 ($SD = 2.88$) improperly indented lines of code per implementation. The mean for the control subjects was 12.7 ($SD = 10.8$). This means that CPP subjects maintained the structure of their code by incorrectly indenting roughly 10 less lines than the control subjects. This difference is statistically significant ($F(1, 11) = 5.61$, $p = 0.0373$) and large (effect size = 0.95).

7.1.6 Discussion

Overall, these results were deemed encouraging enough to move forward with the development of ProPl. Although only marginally significant results were found when counting the number of floundering episodes, with such a small N, it was suggestive that CPP seemed to

[6] Generally, a kappa value above 0.80 is considered reliable.

help them. It is important to note that *only the posttest project was used in the floundering analysis*, and so the CPP's influence seemed to extend beyond the tutored projects. Lack of comments in programs is often a problem for novices, and so the finding that significantly more comments existed in code written by CPP students suggests that the tutoring encouraged them to be more verbose and perhaps implies that they tended to think more about programming in their own words. The mere existence of comments is only a suggestive, however, and so a more detailed analysis of the content of these comments would need to be done to confirm that hypothesis. This was not done as part of the experiment. Finally, it is highly likely the indentation benefits were due to the fact that students were allowed to take the pseudocode solution with them to do the actual program. As stated above, the indentation analysis was done on the Hailstone program since a ceiling effect was detected on the posttest project. Thus, the indentation result is not as strong regarding the longer term effect of CPP.

In sum, the CPP experiment provided a sufficient corpus to analyze tutoring tactics and collect examples of student responses. These data were used in the construction of ProPl as detailed in the previous chapter. The experimental evaluation also provided some valuable insights into the problem of assessing the process of programing and the integration of pre-practice tutoring into a course curriculum.

7.2 PROPL EXPERIMENT

An experiment to evaluate ProPl was conducted in the spring of 2004. It was designed to highlight ProPl's use of natural language dialogue and tutoring tactics and to assess its impact on students' programming experiences, general algorithm writing ability, and program planning skills. Because CPP seemed to be helpful and students received it well, the goals of this experiment were more directed. In particular, it was designed to focus on the benefits of tutoring with natural language dialogue and to test the ability of state-of-the-art dialogue technology to deliver CPP-like instruction.

7.2.1 Design

Participants were randomly assigned to one of two conditions: ProPl or the "click-through" reading of ProPl-C. As described in chapter 6, the differences between the two were minimal, except for the use of dialogue. Students using the control system observed problem decompositions and solution compositions by clicking a button to see the material. The tutorial content was authored such that it mirrored the content presented in the dialogues as much as possible. Both groups viewed the same pseudocode solutions presented in the same staged fashion. The same design notes were also identified and posted. The experiment was designed so that the only real difference between the two groups was the style of interaction.

There are two possible confounds in the design, however. One lies in the difference in content between the mini-lessons and the dialogues. As discussed, this was kept to a minimum by careful authoring of the reading. Secondly, the students using ProPl received a five minute tutorial on the system which presented a simple three line program. In this tutorial, the student does no typing and the problem difficulty is equivalent to that of a program viewed in the second week of class.

7.2.2 Participants

The participants were university students currently enrolled in one of three sections of introductory programming at the University of Pittsburgh (again, CS0 service courses intended for non-majors or potential majors requiring remedial coursework). Two of the sections used Java (those in the Computer Science department) and the third used C (in the Information Science department). All three covered content typical for such courses (types, variables, operators, control structures, functions, arrays, and files). Students were admitted on a voluntary basis and admitted into the study if they had very little or no programming experience before taking the class (no more than one semester). Participants were paid $7/hour for their time.

Out of the roughly 90 total students enrolled in the three courses, 33 volunteered to participate in the study. Of these, 3 were turned away because they had too strong a programming background (over 1 semester prior experience). The initial assignment of students was n=15 to each condition. After attrition, the numbers fell to $n = 12$ and $n = 13$ in the ProPl and ProPl-C groups respectively.

7.2.3 Materials

Just as in the CPP study, the experiment began about one month into the semester and students had covered some basic programming concepts. The study spanned about five weeks, during which students learned control flow (conditionals and loops) and how to write simple subprograms.

7.2.3.1 Class programming projects Instructors in the three courses agreed to give the same two assignments needed for this study (Hailstone and RPS). To help drive home the CPP model, students used the system for a third time for an extra problem that they were not required to solve for their class. This problem, called **Snap** requires that the program allow the user to enter numbers until a user specified number of repeats is detected. The program is to print the mean of the numbers seen prior to the repeating sequence.[7] The full problem statement appears in appendix A.

For students in the Java courses, it was possible to collect online protocols but it was not possible to collect this data from some students in the C section (they worked on their personal computers rather than the central unix server). The numbers for each condition for which online protocols were available are $n = 8$ (control) and $n = 9$ (ProPl). Final solutions to each project were available for all participants. It should also be noted that on-line protocols were available for all students in the posttest programming project (next section) since it was completed in a lab environment.

7.2.3.2 Programming tests Two written programming tests were developed. The first was given as a pretest to gauge students' general incoming programming competence. The test included conceptual questions (multiple choice, true/false), program output questions,

[7] The name comes from the fact that the program should break off a sequence, i.e., "snap" it, once the specified number of repeats has been entered.

and code-writing problems. Students were given 45 minutes to complete this test. Java and C versions of the test were given depending on the course the student was taking.

When students took the pretest, they had only encountered basic topics like types, variables, expressions, i/o, etc. As such, minimal problem solving ability was able to be assessed on this test. The posttest was quite different in that it targeted students planning and algorithm writing skills. It consisted of six distinct questions:

1. Given a "bag" of pseudocode steps and a problem, assemble a solution to the problem using those steps. Not all steps were needed and some duplication was required.
2. Using the same steps, organize them by the goals they help achieve (i.e., plan identification).
3. Read four small code segments and describe the programming goal each achieves as concisely as possible.
4. Given a partially completed program written by a fictitious friend, state the next two goals that should be pursued.
5. Produce a design for the Snap problem (i.e., the third problem that was tutored in the study)
6. Produce a design for the game of Craps (rules are given).

75 minutes was allotted for students to finish, which was sufficient for about 80% of the students in the study. Questions 1 and 2 included some "red herring" steps and some that needed to be used more than once in the solution. Question 3 was a far transfer problem intended to determine if students could reason from actual program code to an abstract statement, in words, of its purpose. Questions 5 and 6 were open-ended design tasks with a restriction on 5 to clearly state a set of programming goals.

In addition to the written posttest, students also completed a *charette* at the end of the study.[8] This assignment, called *Count/Hold*, asked the student to play a simple dice game between opponents. Two players roll dice, and after each roll, the user has the choice to count the roll for that game (getting that score), or hold their roll over to add to the next

[8] A charette is a small programming assignment done under a strict time limit in a closed lab environment. This method of assessment discriminates against students with test anxiety or those who work naturally slower (McCracken et al. 2001).

game (getting a 0 for the current game). The solution requires the implementation of a computer player, allowing the user to play against it, and the play of as many games as the user requests. Students were given two hours to work and were allowed to use their textbook and class notes if desired. The program involved achieving six overlapping goals and required the use of loops, conditionals, and several advanced plans.

7.2.3.3 Survey At the end of the study, students were given a survey consisting of 15 questions (with two extra questions for ProPl students to evaluate the natural language capabilities). These questions targeted students' attitudes about the software and how it impacted them during and after their implementations. Several questions also asked about their use or recollection of the design notes versus the pseudocode. A five-point Likert-type scale (1=strongly disagree to 5=strongly agree) was used to score each statement on the survey.

7.2.4 Procedure

Students completed the following steps in the ProPl experiment (in order):

1. consent form and background questionnaire
2. written pretest
3. tutoring session for Hailstone followed by independent implementation
4. tutoring session for RPS followed by independent implementation
5. tutoring session for Snap
6. written posttest
7. programming posttest (Count/Hold)

For Hailstone and RPS, students were instructed to not start the assignment before coming in for their tutoring session. Students were also *not* allowed to take notes or record the content of their sessions in any form. As discussed earlier, this was permitted in the CPP experiment, and proved to be a problem in the study. To test for direct effects, it was necessary that students be faced with the task of recreating the solution they learned about. In other words, the idea is to force students to solve the composition problem all over again, but after their experimental intervention. Thus, note-taking was outlawed for students in both the ProPl and ProPl-C conditions.

7.2.5 Intention-based scoring

Before presenting the results of the experiment, it is necessary to describe the approach used to evaluate student programming ability. Although final scores can be used to analyze the ultimate success of a student's programming ability, they reveal little about the path taken to that final product. Such scores are best understood as testing for *indirect effects* since there are other potential factors that appear between the time of the intervention and the scoring. Finding indirect effects is highly desirable, but often very difficult to achieve. If no indirect effects are found, the goal is then to find any *direct effects* that may have occurred. In the case of this study, grading a final program is difficult because of many issues, like time and the influence of outside sources (tutor, friends, etc.) on the program produced by the student.

To test for direct effects, the idea of collecting an online protocol (i.e., the collection of all programs submitted to the compiler) was presented in the previous chapter. This collection offers a rare glimpse into the thinking of a novice programmer, but can also be very messy. Compile attempts can be made for a myriad of reasons, some of which are not readily apparent from the textual differences between successive programs. Researchers are just beginning to develop tools for analyzing such data to reveal what it says about programming behaviors (Jadud and Fincher 2003). Simple measures, like raw compile counts and time spent between compiles, are rough and do not necessarily correlate with programming ability. For example, when a student compiles excessively, they may be "tweaking" their code rather than repairing significant bugs. Algorithmic floundering was was proposed as an improvement over such raw measures, but was found to be too difficult to interpret in a precise way. Floundering results say nothing about the kinds of errors being made, and perhaps speaks more to debugging ability and persistence than it does about algorithm writing ability.

It was not possible to perform testing for direct effects in the CPP study since students were allowed to take the pseudocode solutions with them after their tutoring sessions. In other words, programming for students in the study was reduced to a translation task, from pseudocode to their target language. This meant they were not required to solve the composition problem again. This mistake in the design of the CPP experiment that was not apparent until far too late, but was rectified in the PROPL experiment.

In this section, a new way of deriving interpretable scores from online protocols is developed, namely, *intention-based scoring* (IBS). The goal is to provide a score that more accurately assesses a student's ability to solve the composition problem, and therefore assess direct effects. IBS derives elements from previous work on identifying bugs in online protocols including intention-based diagnosis from PROUST (Johnson 1990), as well as on establishing cognitively plausible accounts for how novice bugs are produced (Spohrer, Soloway, and Pope 1989). The remainder of this section provides a detailed outline of three steps necessary to perform IBS: inspection of an online protocol, bug identification, and scoring.

7.2.5.1 Inspecting an online protocol The first step in producing an IBS is to identify the subset of programs from an online protocol to analyze for bugs. As discussed above, this subset of programs should be the student's *initial attempts* at achieving each goal. A judge begins with the first program submitted and works through the protocol chronologically, checking off goals along the way. This stage is complete when attempts at all goals have been identified or the protocol ends (leaving some goals un-attempted). The process of protocol subset identification is depicted in the top half of figure 7.2.

The identification process is not always straightforward, however. The first program in a protocol is not always a legitimate goal attempt, for example. Some novices prefer to compile very simple programs to begin, including only things like variable declarations or simple print statements. Cases like this are ignored and the search continues sequentially for the first substantive attempt at achieving a goal. A related issue is the sometimes fuzzy question of whether or not a program represents an attempt at achieving a goal or not. It is not quite as simple as saying "if any plan component is present, then count it as a goal attempt." For example, some students prefer to declare and initialize all variables at once. This certainly does not imply the student is attempting to implement all plans in which these steps participate.

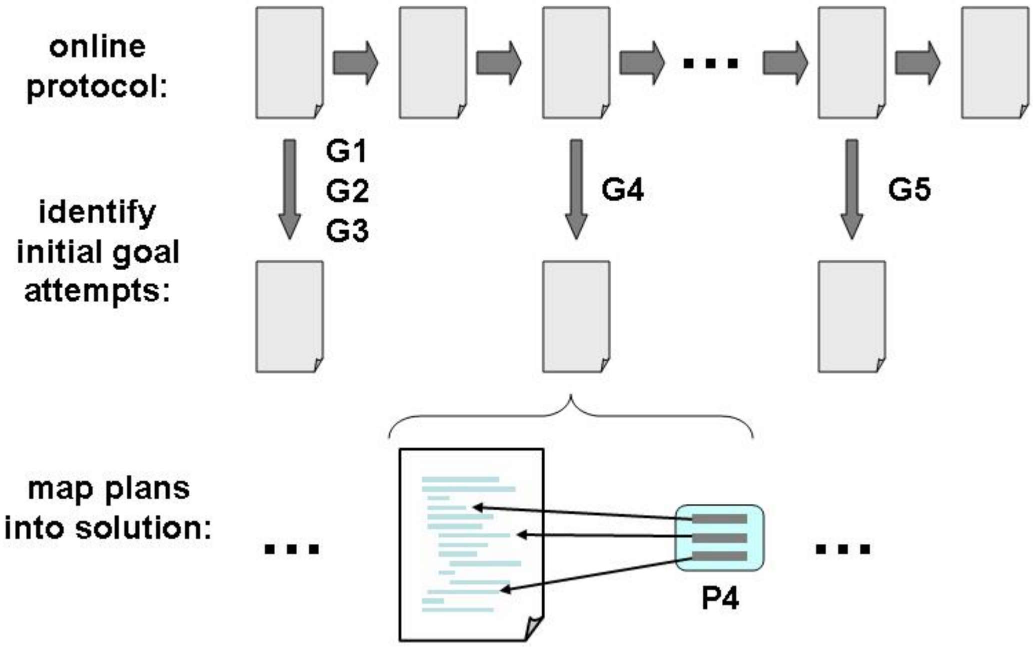


Figure 7.2: The first two stages of computing an intention-based score. From a protocol, the programs representing the first attempts at goals are identified, then these are matched to known correct plans that can achieve them.

To handle problems like this, the method of selecting programs throughout a protocol must be agreed upon between multiple judges. In the example below, the consensus with the variable declaration issue, for example, was to conclude that by itself, a declaration would not be counted as an attempt at its plan. In other words, more plan components would need to be present than just a declaration or initialization step to count as an outright attempt at that goal. Such issues arise frequently in the subjective tagging of data, which is why it is recommended to tag some subset of the data together under open discussion.

7.2.5.2 Bug identification A critical component of IBS involves the identification and classification of bugs present in a student's protocol. The same two stage process is followed which is described in (Spohrer, Soloway, and Pope 1989). First, the plans being implemented by the student must be identified, and next, compared to the known correct plans of an implementation. Bugs then fall out as differences between these two structures. For IBS, of course, only the plans corresponding to the new goals being implemented at each stage should be considered.

Although there are certainly many ways to characterize the bugs (i.e., plan differences), a simplification of the approach taken in (Spohrer, Soloway, and Pope 1989) is adopted here. Most generally, an IBS scheme could be constructed from any similar bug classification strategy. Because the goal here is not to provide an account of cognitive plausibility, categories that relate to solving the composition problem are the only ones used. The top-level categories of bugs in the coding scheme are:

- **omission**: A plan component is missing.
- **malformation**: A component is incorrectly implemented.
- **arrangement error**: A component was placed in the wrong location.

In addition, when inspecting a program, it is also necessary to identify those bugs that are a result of *merging* of plans (e.g., the multiple loop issue mentioned above). Bugs that are *not* a result of confusion between multiple plans are referred to as *isolated*. Of course, some bugs can fall under multiple categories. For example, a step can be malformed, out of place, and be a result of confusion between two plans. Because this is a subjective tagging process, it is recommended that multiple judges be used and agreement be checked.

An example of bug identification is shown in figure 7.3. In this example, the student is attempting to implement a counter plan (shown in the shaded box), but has made three mistakes. First, the incorrect value is used for the initialization step (it should be 1). Second, the increment step is not placed inside the loop body (an arrangement bug). Finally, there is no print statement (a bug of omission). In this case, the arrangement bug is also considered a plan-merging error since the counter is being integrated into the looping code, which was already in place to achieve a different goal in this problem.

7.2.5.3 Scoring With a bug-tagged protocol, the final step in computing an IBS is to apply a scoring rubric. Although simple bug frequencies could be used, it is less fair since focal steps in plans (i.e., those that are "more" central, such as the increment step of a counter

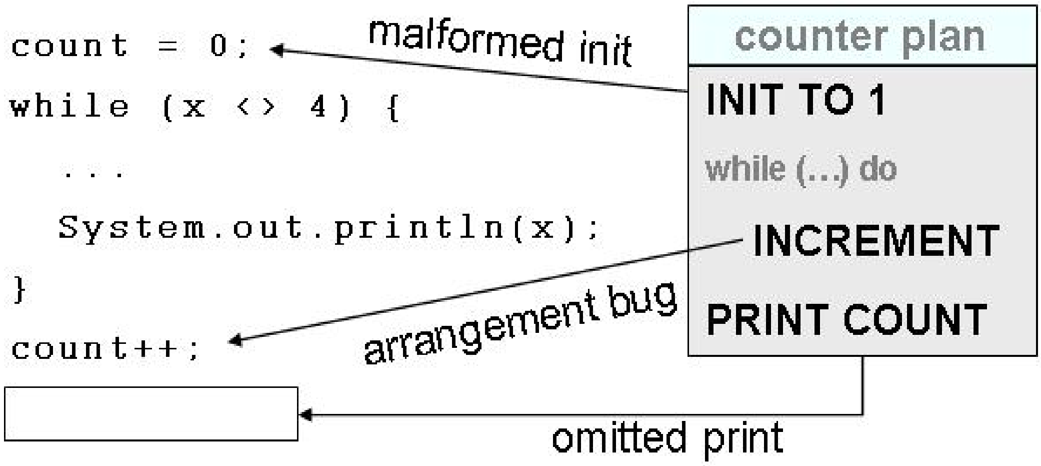


Figure 7.3: To find bugs via plan differences, the Hailstone counter plan (on the right) is mapped into the student's code. Three differencs fall out of this comparison, including a malformed initialization step, an incorrectly arranged counter step, and finally an omitted print statement.

plan) would count the same as less critical components (such as an output statement). To create a rubric, points need to be assigned to the various plan components. Focal steps are weighted more heavily, like updates and conditions, than are supporting steps, like initializations and output statements. This allows discounting of slips (like forgetting to print a value) and the highlighting of errors in critical plan steps. Finally, points for each bug identified during the analysis are taken away from an overall possible score, thereby producing a final intention-based score. In sum, this score represents the accuracy of students' first attempts at achieving programming goals. By looking at points lost from each of the subcategories (like merge errors or omissions), one can get a better feel for the kinds of errors novices produce.

As an example, for the counter plan in figure 7.3, one possible assignment could be 3 points for the initialization step, 5 for the increment step, and 2 for the print step. As mentioned, it is best to perform this stage with expert instructors who have experience creating rubrics. In the example, the student would lose 1 point for the incorrect initial value, 3 for the improper location of the increment step, and 2 for forgetting the print step. These partial values need to be agreed upon in the rubric. In sum, this student would receive 4 out of 10 possible points for this attempt at implementing a counter plan.

7.2.5.4 IBS Discussion One difference of IBS with previous work using online protocols is that *all* attempts in a protocol are made available for inspection. Most previous work considered only syntactically correct compile attempts. The reasoning behind "opening" up the protocols in this way comes from the observation that for some students in the protocols, the algorithm intended by the first compile attempt was often different than that in the first syntactically correct attempt. This means that students' algorithms seemed to change, likely inadvertently, while fixing syntax errors. The very first attempt at assembling an algorithm is likely a more accurate representation of a student's initial impression at how to solve the composition problem. Also, the process is dubbed "intention-based" for two reasons: first, programs are inspected by inferring what goals the student is trying to achieve. Second, when a program statement is not syntactically correct, it is necessary to infer what plan component is being attempted. The line `count + 1;`, for example, is likely an attempt to increment a counter variable. Thus, such a statement is considered equally as correct as a syntactic one.

There are several problems with the IBS method of evaluating programs. First, it is extremely tedious. In no way is IBS intended for regular classroom evaluations – it is only reasonable for use in targeted evaluations that require a fine-grained understanding of student success. In fact, it would be largely unfair to use IBS for the purposes of assigning a grade since debugging, optimization, etc. are all important skills as well. Second, the creation of the rubric is subject to the bias of the researcher. In other words, the weighting of the various plan components may indirectly impact the outcome of the study. To deal with this problem, the *raw frequences* are discussed (section 7.3.2.4 and appendix C), which are immune to bias potential (but fail to produce an accurate assessment). Finally, in the form presented here, IBS is dependent on a plan-based theory of programming knowledge. The difficulties and limitations of this theory, such as handling the notion of recursion, are naturally inherited.

7.3 RESULTS

The primary objective of this evaluation was to test the efficacy of natural language tutoring to teach simple program planning skills. The hypothesis was that students who learn decomposition and composition skills in the context of their class assignments will learn such skills more effectively if they are engaged in natural language dialogue as opposed to reading the same material. In this section, results from the programming assignments, posttests, and follow-up survey are reported. For the programming assignments, baseline data is included from students who received no tutoring or intervention at all. These students used Pascal and the data was collected from the CPP study.[9] By looking at the performance of students who receive no special intervention, a better idea of where ProPl and ProPl-C sit with respect to students who learn in the traditional way can be formed.

7.3.1 Pretest

On the pretest covering programming competence, scores were similar for the ProPl group ($M = 68.9$, $SD = 19.2$) and control (ProPl-C) group ($M = 67.5$, $SD = 18.7$), $t(24) = .155$, $p = .88$. This indicates no difference in incoming programming ability allowing the conclusion that the assignment to conditions was fair with respect to incoming student ability. Because the students in the baseline group were not given this pretest, it was not able to include them in the above test of equivalence between groups.[10] Thus, in the sections below, t-tests were used for comparisons involving the baseline group, whereas ANCOVAs were used for comparisons between the control and ProPl groups to factor out the pretest.

[9] Two more protocols from untutored students became available after the CPP study, taking the n for the baseline group to 9.

[10] However, looking at the final grades students in the three groups received in their respective introductory programming courses suggests roughly equal competence. The GPAs were 2.89, 2.88, and 2.89 for the baseline, control, and ProPl groups, respectively. In addition, each group had a roughly even split of A, B, and C students. Although weak evidence, it is at least suggestive of equivalence between the three groups.

Table 7.1: Final program means and standard deviations in parentheses using traditional scoring rubrics that measured execution behavior, quality of code, and style. All scores are out of a possible 100.

problem	baseline	ctrl	ProPl
Hailstone	86.0 (13.4)	84.3 (19.3)	90.1 (11.4)
RPS	78.8 (23.0)	76.3 (22.9)	75.8 (23.5)
CH	n/a	45.6 (23.5)	48.3 (32.3)

7.3.2 Programming projects

The programming data analyzed in this study covered three programming projects: Hailstone, Rock-Paper-Scissors (both described in section 7.2.3.1), and Count/Hold (described in section 7.2.3.2). Count/Hold was a *charette*: students worked in the lab under a two hour time limit and were not given any help (other than with technical difficulties). In this section, we first report final scores on these projects followed by the intention-based results.

7.3.2.1 Final program scores The final scores for the three programming projects involved in this study are shown in table 7.1. The control and ProPl groups received tutoring for the Hailstone and RPS projects as discussed in the previous section. Students in the baseline group attempted the programs with no intervention at all. Programs were graded independently by two experienced instructors using rubrics similar to those used in other studies (e.g., McCracken et al. 2001) and agreement between the graders over all programs was very high ($r(83) = .852$, $p < .0001$). None of differences between groups were found to be statistically significant.

In understanding why no differences were found between the groups using the final program scores, it is useful to draw the distinction between *direct* and *indirect* effects. A direct effect is one that is most closely related (both chronologically and conceptually) to the intervention. Final program scores do *not* fall into the direct effect category because of (1) the

Table 7.2: Composite intention-based are derived from a goal/plan analysis of students' online protocols and represent a composite score of the accuracy of students' first attempts at achieving programming goals. All scores are out of a possible 100.

problem	baseline	ctrl	ProPl
Hailstone	69.3 (16.4)	79.8 (15.4)	86.1 (9.46)
RPS	67.7 (22.5)	59.5 (18.7)	77.5 (16.4)
CH	n/a	49.1 (26.3)	64.1 (29.8)

large number of potential influences (such as friends, TAs, etc.) and (2) the introduction of other domain factors, such as confusion over syntax, debugging skill, and extended time available to continue working. In short, final program scores are indirect effects because of these complications. In this light, it is not surprising to find no differences between these indirect measures. To get at a more direct measure, it is necessary to turn to intention-based scores.

7.3.2.2 Composite intention-based scores As discussed in section 7.2.5, intention-based scores are designed to reveal more about the students' process by measuring the correctness of their first attempts at each programming goal (see section 7.2.5). The reader is reminded that because there was no access to online protocols for all subjects, the number of subjects is reduced for the Hailstone and RPS projects. The respective n's for the baseline, control and ProPl groups were 9, 8, and 9. The protocols for these students were coded for bugs to produce the intention-based scores by two experienced instructors. After training together on roughly 15% of all programs and scoring another 20% independently, a kappa was again computed to assess agreement. In this case, the tags for all plans were compared between graders. The result was a kappa of .865 which indicates very high agreement[11]

[11] Such a high agreement is not as surprising as it may first seem. For example, it is easy to see when a step is arranged improperly or when a statement is malformed. Disagreements, when they did occur, generally involved the classification of a malformed step as a merging error or on the intentional interpretation of a step (i.e., deciding on the intention underlying a syntactically wrong step).

The intention-based scores are shown in table 7.2. Some significant and marginally significant differences exist between the groups. Considering first how the baseline group compared with each of the other groups, for Hailstone, the ProPl students ($M = 86.1$, $SD = 9.46$) outscored baseline students ($M = 69.3$, $SD = 16.4$). This difference is statistically significant ($t(16) = 2.12$, $p = .0017$) with a very large effect size ($es = 1.03$). The control group also outperformed the baseline group on Hailstone, but not significantly. On the RPS problem, the baseline group ($M = 67.7$, $SD = 22.5$) actually outperformed the control group ($M = 59.5$, $SD = 18.7$), but the difference is not significant. ProPl students ($M = 77.5$, $SD = 16.4$) did outperform the baseline students, but again, not to a significant level. Because the baseline students were not a part of this study, they did not do the Count/Hold problem.

Turning now to the ProPl and control groups, all students in these groups took the pretest described in section 7.2.3.2, and so ANCOVAs are used for statistical tests in order to factor out pretest performance. Although ProPl students outperformed the control students on each project, the only significant difference is on RPS. ProPl students ($M = 77.5$, $SD = 16.4$) were significantly better than control subjects ($M = 69.5$, $SD = 18.7$), $F(1, 15) = 7.88$, $p = 0.015$, $es = .96$. On Count/Hold (the untutored posttest charette), ProPl students ($M = 64.1$, $SD = 29.8$) outperformed those in the control group ($M = 49.1$, $SD = 26.3$) to a marginally significant level ($F(1, 22) = 3.59$, $p = .072$, $es = .57$).

7.3.2.3 Decomposed intention-based scores The results shown in table 7.2 are composite scores; that is, the various categories of bugs (discussed in section 7.2.5) are lumped together to produce the overall score. To reveal how these points were distributed across the various bug categories, the composite scores were broken down according to several categories:

- **Merge errors:** points lost due to the interaction between separate plans in a solutions (includes arrangement and some step malformation errors).
- **Plan component omissions:** points from missing plan parts.
- **Remaining isolated errors:** non-merge related malformation errors of individual plan components.

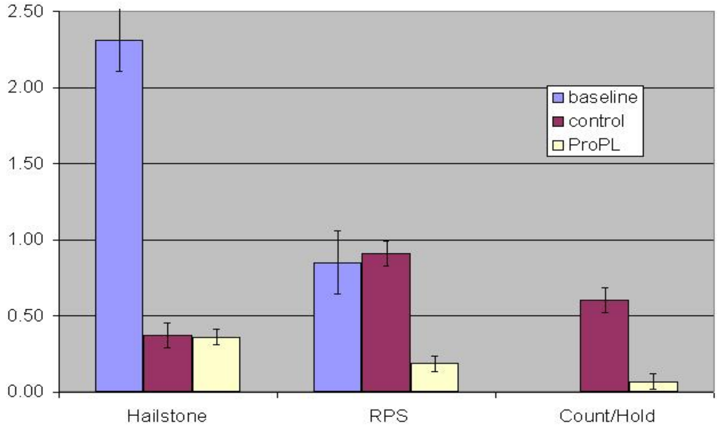


Figure 7.4: Merging related points can result from step arrangement errors or malformed statements that involve aspects of two distinct plans. As before, points lost are shown per opportunity to make an error along with standard error bars.

It would be misleading to use the total points missed for each of these categories. For example, a student who attempts two goals out of a possible five would have far fewer opportunities to produce merging errors than someone who attempts to solve all five. The resulting merge error score would be deceptively low. Similar arguments can be made for plan component omissions and isolated errors. Rather than using raw points, then, the solution is normalize and compare the points lost *per opportunity to commit that error*. For the error categories identified above, the *total number of goals attempted* is used as a denominator.[12]

Figure 7.4 shows the points lost from merging related errors over the three programming problems. For the Hailstone problem, the control group (M = .38, SD = .65) produced significantly fewer merging related errors than the baseline group (M = 2.31, SD = 1.9), $t(15) = 2.76$, $p = 0.015$, $es = 1.0$. The ProPl group (M = .36, SD = .45) performed similarly well when compared to the baseline group ($t(16) = 3.02$, $p = .008$, $es = 1.0$). On RPS, the ProPl group (M = .19, SD = .21) outperformed both the baseline group (M = .85, SD = 1.0) to a marginally significant level ($t(15) = 1.98$, $p = .075$, $es = .66$) as

[12] For merge errors, total number of attempted goals -1 was used because at least two plans are required for a merge error to be possible.

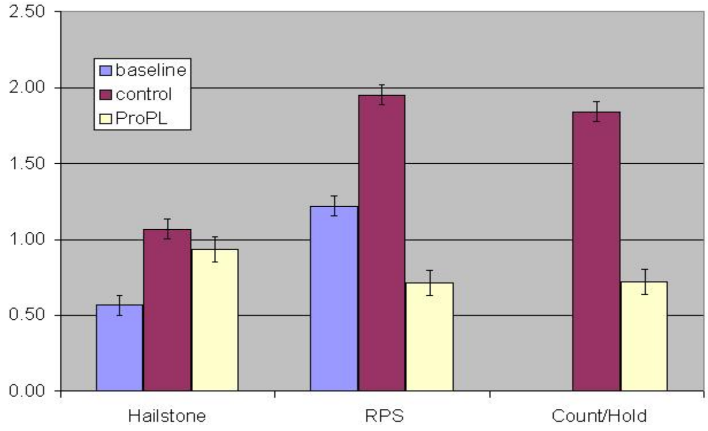


Figure 7.5: Plan part omission points lost per plan implementation attempt.

well as the control group ($M = .91$, $SD = 1.1$), $F(1, 15) = 3.71$, $p = .076$, $es = .65$. Finally, on the Count/Hold project, the ProPl group ($M = .07$, $SD = .24$) again surpassed the control group ($M = .61$, $SD = .74$) but this time to a significant level ($F(1, 15) = 5.77$, $p = .026$) and with an extremely large effect size ($es = 2.3$).

Next, looking at students' ability to produce complete plans (figure 7.5), several differences were found to be significant. Interestingly, for the Hailstone problem, the baseline group lost *fewer* points for missing plan parts ($M = .57$, $SD = .59$) than the control group ($M = 1.1$, $SD = .53$) to a marginally significant level ($t(16) = -1.85$, $p = .085$, $es = 1.0$). When compared to the ProPl group, the difference is not significant. With more points lost in the merging category by the baseline group, however, losing fewer points in the omission category makes sense. On RPS, the baseline group ($M = 1.22$, $SD = .62$) again outperformed the control group ($M = 1.95$, $SD = .67$), but to a significant level ($t(16) = -2.20$, $p = .046$, $es = 1.2$). The control group, in general, seemed to be more forgetful than the other two groups. The ProPl group ($M = .71$, $SD = .63$) also was significantly better than the baseline group on RPS ($F(1, 15) = 15.6$, $p = .0017$, $es = 1.9$). A similar difference appeared on Count/Hold with the ProPl group ($M = .72$, $SD = .62$) losing significantly less than the control group ($M = 1.84$, $SD = 1.13$), $F(1, 15) = 9.22$, $p = .0065$, $es = .99$.

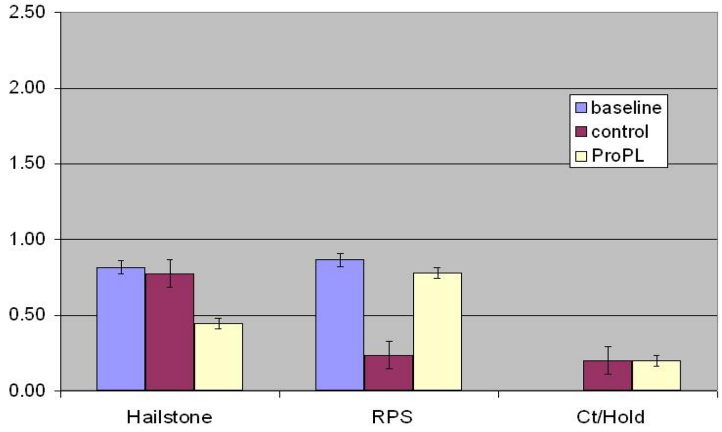


Figure 7.6: Points lost for isolated (non-merging) errors, per plan implementation attempt

Lastly, turning to isolated errors, which are essentially all malformations not related to the complications of plan merging, very few differences exist between the groups (see figure 7.6). In fact, the only statistically significant difference occurred in the Hailstone problem between the ProPl group ($M = .44$, $SD = .31$) and the baseline group ($M = .82$, $SD = .4$), $t(16) = 2.22$, $p = .042$, $es = .95$. Although the control group ($M = .24$, $SD = .63$) did lose fewer points per attempt than the ProPl group ($M = .78$, $SD = .70$) on RPS, the difference is not significant ($F(1, 15) = 2.03$, $p = .18$). The absence of isolated errors in RPS by control students does make sense when compared to their higher rate of omission errors; i.e., they had fewer chances to make isolated errors because they implemented fewer plan components overall.

7.3.2.4 Bug frequencies There is a potential bias involved in the creation of any rubric. In the case of IBSs, the point values assigned to each plan component and how points are taken away for plan differences are susceptible to such a bias. To confirm that the rubric-based scores were indeed representative of the bugs underlying them, bug frequencies were collected from the original IBS codings and counted. Bug frequency counts are best thought of as a rubric that assigns one point to each plan part, rather than weighting various pieces differently. This removes the chance for bias since all plan components are counted equally. In addition, there is no opportunity for partial point deductions, which is another potential source of rubric bias.

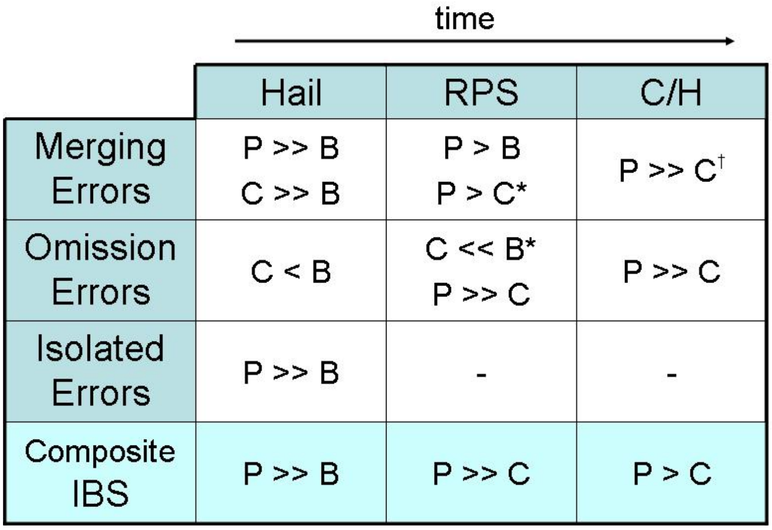
Because the results were very similar to the rubric-derived scores, the full results are not shown here (but do appear in appendix C). The graphs for each of the categories (merging errors, omissions, and isolated errors) all reveal the same relationships between the various groups and projects. Even though, some of the statistical results did vary:

1. For merging errors on RPS, the ProPl group difference was found to be significant over the baseline group (rather than marginal).
2. Again for merging errors on RPS, the ProPl group difference over the control group was not significant (rather than marginal).
3. For merging errors in Count/Hold, the ProPl group difference was found to be marginally significant (down from significant).
4. For omissions in Hailstone, the baseline group difference over control was found to be significant (rather than marginal).
5. Also with omissions in Hailstone, a marginal difference favoring the baseline group was found over the ProPl group (no difference with the rubric).
6. For omissions on RPS, the significant difference favoring the control group over the baseline did not appear with bug frequencies.

Out of the 10 significant or marginally significant differences found using the rubric, seven of them either stayed the same or were stronger using bug frequencies. Only one was weakened from significant to marginal, and two differences did not appear. Since the *basic relationships between the groups were maintained* and *no drastic statistical shifts in either direction occurred*, the suggestion is that the original rubrics used were fair.

So why not use only the bug frequencies? A large price is paid when looking only at frequencies. The drawback is that all bugs contribute equally to the overall score. This means that, for example, misplacing a printing step in a counter plan (a common slip) is treated the same as misplacing the focal increment step (which is clearly more important). In other words, less important steps in plans influence the overall score too much, which is

Table 7.3: A summary of all IBS results. **P** represents the ProPl group, **C** the control, and **B** the baseline. A double relational sign (e.g., >>) implies significance and a single implies marginal significance. Note that these refer to *performance* (e.g., $P >> B$ means P outperformed B. † and ∗ indicate that signifance was reduced in the bug frequency analysis to marginal and none, respectively.



	Hail	RPS	C/H
Merging Errors	P >> B C >> B	P > B P > C*	P >> C†
Omission Errors	C < B	C << B* P >> C	P >> C
Isolated Errors	P >> B	-	-
Composite IBS	P >> B	P >> C	P > C

also the probable explanation for many of the differences described above. Errors on critical plan components *should* influence scores more heavily, and so the focus in the discussion in the next section is primarily based on the rubric-based scores.

7.3.2.5 Summary of IBS results Decomposing the IBS results has the unfortunate side-effect of greatly muddying the overall evaluation picture. To help bring all of these results together, all significant and marginally significant results are shown in figure 7.3. Significant relationships show a double-greater or less-than sign, while marginal differences use single. Also, the relationship refers to *performance*, not scores. So, for example, ProPl students outperformed control students on the Count/Hold project by losing fewer merging related points per plan merging attempt.

Most cells show that the relationships run in the expected directions. The only cells where the differences run contrary to expectations occur in the omission row. The control students were clearly more forgetful than the baseline group in both RPS and Hailstone. However, the price for this is revealed by their poor showing in the merging row, which shows consistent baseline shortcomings. Possible explanations for this is given below, in section 7.3.6.

7.3.3 Written posttest

The written posttests were graded by three experienced programming instructors. Each of the six questions was scored on a scale of 0 to 5 (0 = no attempt, 5 = excellent) giving a maximum possible score of 30. The first five tests were graded together to refine the grading rubrics. Agreement on the remaining exams between the graders, as calculated by a linear regression, was extremely high for each pair ($p_{1,2} = .95$, $p_{2,3} = .94$, $p_{1,3} = .91$, all $r(19) < .001$). To calculate student scores, the mean score of the three graders was computed.

Overall, students in the ProPl group ($M = 19.3$, $SD = 4.38$) scored higher than those in the control group ($M = 17.4$, $SD = 6.14$), but this difference is not statistically significant ($F(1, 23) = 2.04$, $p = .17$). However, because question 3 was a code comprehension question and the only one that does *not require* any program planning skills, it is worthwhile to in consider the overall score without it. The question asked students to read Java or C code and describe the goal it achieved, in a few words. Not surprisingly, most students answered with line-by-line descriptions of how the code operates (e.g., "first a variable is declared, then it is assigned 0, then...") rather than summarizing and abstracting from it as the problem asked. Omitting question 3, ProPl students ($M = 17.1$, $SD = 3.75$) again outperformed those in the control group ($M = 15.1$, $SD = 5.1$), but to a marginally significant level ($F(1, 23) = 3.00$, $p = .097$, $es = .392$).

The questions on the posttest (described in section 7.2.3.2) tested different aspects of the decomposition and composition problems:

- Question 1 tested composition skills.
- Question 2 targeted novices ability to organize code by plans.
- Question 4 involved identifying goals to pursue given code.
- Questions 5 and 6 were open-ended decomposition problems.

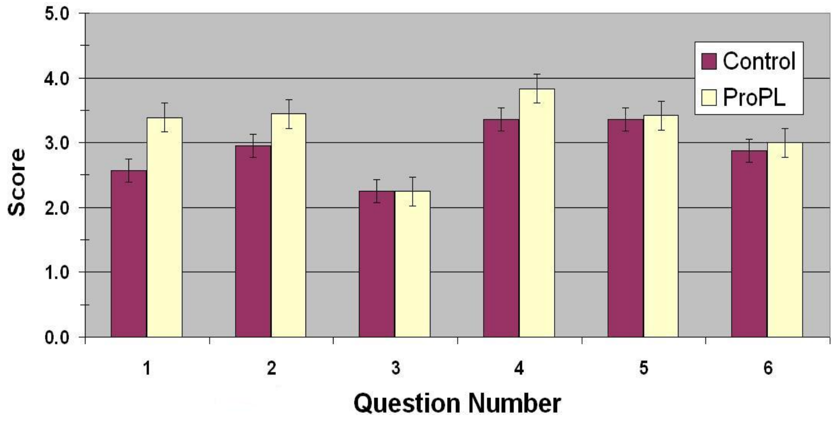


Figure 7.7: Student performance on the written posttest by question with standard error bars shown. Differences in questions 1 and 2 are statistically significant.

Because of this, it is illuminating to view student performance by question rather than overall performance. Such a breakdown is shown in figure 7.7. On question 1, ProPl students ($M = 3.39$, $SD = 1.2$) scored higher than control students ($M = 2.56$, $SD = 1.4$), which is a significant difference ($F(1, 23) = 5.72$, $p = .026$, $es = .61$). On question 2, ProPl students ($M = 3.44$, $SD = .70$) similarly were better than control students ($M = 2.95$, $SD = .84$) to a significant level ($F(1, 23) = 24.6$, $p < .001$, $es = .58$). ProPl students ($M = 3.83$, $SD = 1.1$) scored nearly a half a point higher than the control students on question 4 ($M = 3.36$, $SD = 1.2$), but the difference was not found to be significant ($F(1, 23) = 1.0$, $p = .33$). None of the differences on the remaining questions approach significance.

7.3.4 Survey results

Of the 15 questions both groups received on the follow-up survey, four resulted in statistically significant differences between the ProPl and control groups. These are shown in table 7.4 along with two other items of note. Based on these responses, it seems that ProPl students believed they understood the material less, focused on the design notes to a greater degree than pseudocode, found debugging easier "because of the use of the system," and that they

Table 7.4: A subset of the survey results comparing mean evaluation scores (1 = strongly disagree, 5 = strongly agree) of students who used ProPl against students who used ProPl-C. * indicates statistical significance ($p < .05$) while † implies marginal significance ($p < .1$), as computed by 2-tailed t-tests.

survey item	Ctrl (SD)	ProPl (SD)
Would use system again on my own (w/no pay)	4.2 (.94)	4.7 (.65)
I understood the explanations given*	4.7 (.49)	4.1 (.79)
I tried to remember the design notes*	4.0 (1.0)	4.8 (.39)
I tried to remember pseudocode	4.8 (.62)	4.5 (.52)
Debugging was easier because of software*	3.3 (1.1)	4.4 (.79)
I had influence on the pseudocode†	2.8 (1.3)	3.7 (.89)

had more influence on the pseudocode as it was being constructed during tutoring than students in the control group. In reality, neither group had any influence on the pseudocode as it was hand-authored ahead of time. Students in both groups indicated a desire to use the system again in the future, for their own benefit and with no payment, but the difference was not significant between groups (first line, table 7.4). There is a possible ceiling effect with this question due to the natural bias of using volunteers as participants.

Looking now within the groups, significant differences are found in questions that reveal how the content of the tutoring was perceived by the students. Two items on the survey asked to what degree they tried to remember the design notes and the pseudocode after their sessions. Within the control group only, students claimed they tried to remember the pseudocode ($M = 4.8$, $SD = .62$) more than they tried to remember the design notes ($M = 4.0$, $SD = 1.0$), $t(22) = -2.14$, $p = .044$. Checking the same questions within the ProPl group, the inverse of these rankings was observed. That is, pseudocode ($M = 4.5$, $SD = .52$) received a lower ranking than design notes ($M = 4.8$, $SD = .39$), but only to a marginally significant level ($t(22) = 1.77$, $p = .090$).

7.3.5 The “I don’t know” crutch

Students are told in the ProPl tutorial that “I don’t know” is an option when they are unable to answer a question. In analysis of the transcripts of students using ProPl it was found that the use of “I don’t know” (IDK) is a reliable predictor of posttest performance. In a multiple regression, the partial (negative) correlation between IDK and posttest was marginally significant ($r^2(11) = .29$, $p = .073$) with pretest score as the other independent variable.

7.3.6 Discussion

First, because of small sizes of the groups in this study, it can be argued that all of the detected differences in the previous section are encouraging, even when full significance was not reached. Nonetheless, these results do point to several findings. Perhaps most importantly, students who were tutored by ProPl *demonstrated enhanced skill at solving the composition problem.* This is supported by several results: ProPl students...

- had consistently higher intention-based scores than students in the other two groups.
- made fewer merging-related errors than students in the other two groups.
- omitted fewer plan parts than students in the control group.
- scored higher on a written algorithm assembly problem than the control group.

Each of these represent important aspects of solving the composition problem. Since ProPl students outperformed students who received no tutoring (baseline group) and students who read the same content (control group), these differences suggest that the dialogue-based interaction led to deeper learning and skill at solving the composition problem.

Turning now to students’ ability to work with plans, the data suggests that ProPl *students adopted a more expert-like view of programming by working at the level of schemata and plans* rather than by the line-by-line, more localized perspective typically taken by novices. This is supported by three results from the previous section. First, fewer merging related errors by ProPl students suggests that they developed a heightened sense of the relationships and distinctions between plans and plan parts. For example, an understanding that two plans should not interact would translate into fewer spurious connections in program code between them. Second, fewer plan part omissions means ProPl students were able to produce more complete plans on their first attempts at implementing them. Third, ProPl students received higher scores on the written posttest problem that asked them to organize steps according to the goals they achieved and the plans to which they belonged. When considered together, these differences suggest that dialogue-based tutoring accelerates the development of the tacit knowledge of programming and the expert-like perspective that code can be viewed in "chunks" dispersed throughout a program.

Looking at how the students perceived the help provided by the two systems, several differences were found (section 7.3.4). For one, *students who used the control system believed they understood the material better than students who used* ProPl. This phenomenon is known as "the illusion of knowing" (Glenberg, Wilkinson, and Epstein 1982) and it seems that dialogue-based tutoring mitigates this effect to a certain degree. It is likely that participation in dialogue is responsible for this difference. In a dialogue context, students often receive negative feedback, and this may help them develop more accurate self-assessment skills. In the control condition, students did not have this opportunity unless they took it for themselves (i.e., unless they self-explained on their own). It is also possible that the subtleties of the problems were exposed in a more memorable way thanks to dialogue.

Another result from the surveys was that ProPl *students preferred the more abstract, conversational-style representation of programming knowledge instead of the more concrete, pseudocode representation.* They rated the value of design notes higher than pseudocode when asked what they remembered when thinking back to the tutoring sessions. The inverse rankings were given by control students, pointing to the pseudocode as the preferred representation. Since design notes consist only of short phrases and sentences, it is a more abstract and less structured representation of the solution. In fact, the intent behind using design notes in the first place was to aid in the reification of the tacit knowledge that underlies programming (i.e., goals and schemata). It seems that the use of dialogue raised the comfort levels of those students to use the more abstract representation, at least in their own estimation. This result is in concert with the previously mentioned result that ProPl students seemed to work more on the plan-level rather than line-by-line.

Performance on the open-ended design questions on the written posttest was surprising. Question 5 asked students to describe their approach to the Snap problem, the third problem covered in the experiment. Question 6 presented a novel problem, Craps, and asked students to produce a set of design notes (i.e., a goal decomposition and ideas about how to achieve those goals). The initial hypothesis was that because ProPl's tutoring posed the "what" and "how" questions, and gave feedback, this practice would lead to improved ability to perform the same task without the guidance of the system. Since control students never generated any verbal descriptions at all, it was surprising to see their verbal abilities on par with the ProPl students (questions 3, 5, and 6 in particular, described in section 7.2.3.2). One possible explanation is that students were not directly responsible for the writing or organization of the design notes, in either condition. Therefore, no students had any direct practice producing decompositions written in natural language. This suggests students should be more involved with creating the design notes and if possible, responsible for the content.

7.4 ANALYSIS OF KCD-DRIVEN DIALOGUES

Recall from 6.3 that ProPl is an application of the Atlas dialogue management system. Pre-authored, hierarchical structures called *Knowledge Construction Dialogues* (KCDs) are used to guide dialogue. To author these knowledge sources, a domain author must produce the tutor's contributions, questions, and sets of expected student answers. In order for the resulting dialogues to be realistic, the authoring of the KCDs needs to be carefully constructed and expected answer lists sufficiently broad.

To provide support that ProPl was able to carry out reasonable dialogues, a *manipulation check* was performed on a random sampling of 100 question/answer pairs from the ProPl corpus. Such an analysis involves asking two questions of each pair:

- Was the answer category selected by Atlas the same that a human judge[13] would select, *from the available categories*?

[13] The author served as judge in this analysis.

- Does the student's answer represent a new category of answers not included in the corresponding KCD?

A manipulation check is therefore intended to reveal whether that the student's answers are being classified properly and that the coverage of answers in the KCDs is broad enough. It should also be noted that the two questions are independent. It is possible that even when the system and human judge's category selections match, that a new category might be needed to properly handle the student's answer.

To select the question/answer pairs from the ProPl corpus, random number generators were used. These were used to first select a problem (Hailstone, RPS, or Snap), then a student id, and lastly an utterance number from the corresponding dialogue. If the utterance selected was not part of a question/answer pair, then another random utterance was selected until an appropriate one was found.[14] The results were positive:

- The system chose the "best" category 84% of the time.
- A new category of answer was needed 18% of the time.

Of the 16% incorrectly classified answers, simply extending the synonym list would have sufficed 75% of the time. In addition, within the 84% of the matches with the human judge, 21% of them occured as an "anything else" match, meaning that the default response was ideal given the student's answer. The remaining 79% of the correctly matched answers were all expected student answers and therefore, responded to as intended by the KCD author. These results are suggestive that student utterances were generally understood and that the KCDs reached an adequate level of robustness.

7.5 CHAPTER SUMMARY

This chapter presented the evaluation of CPP when compared against a baseline group of students who receive no tutoring at all, and an evaluation of ProPl which compared it with ProPl-C, the non-dialogue version of the system. The main points of the chapter were:

[14] Other utterance types included didactic tutor utterances, administrative utterances, and student interface actions.

- Floundering happens when students compile to fix an algorithm-related bug and make no progress in fixing that bug. To identify floundering, one must analyze a student's online protocol (i.e., the collection of all programs submitted to a compiler) and use multiple expert programmers to tag compile attempts as floundering or not.
- Students receiving CPP floundered less than students who received no tutoring on a follow-up, untutored project, but only to a marginally significant level (suggesting there was some longer-term impact).
- ProPl (dialogue) students demonstrated superior performance at solving the composition problem when compared to ProPl-C (read-only) students on various measures.
- Also, ProPl students showed evidence of adopting a more expert-like view of programming by viewing programs by chunks rather than line-by-line.
- ProPl-C students believed they understood the material better than ProPl students, suggesting that dialogue had mitigating effect on the "illusion of knowing."
- ProPl students preferred the design notes over the pseudocode when thinking about their tutoring sessions, suggesting they preferred the more abstract representation of the tacit knowledge of programming.
- ProPl-C students, on the other hand, preferred the more concrete, pseudocode representation.
- A manipulation check performed on the ProPl corpus revealed that the Atlas system chose the best category for a student answer 84% of the time, and that unexpected answers deserving of a new category altogether occurred only 18% of the time.

8.0 CONCLUSION

8.1 SUMMARY

In this dissertation, a model of tutoring for novice programmers called *Coached Program Planning* (CPP) was developed. A central aim of CPP is to model and support the cognitive problem solving and planning activities that novices are known to overlook or bypass altogether. A salient reason why novices are unable to plan effectively is that they lack the experience and generalized programming knowledge that experts possess. As such, a second goal of this work was to explore how natural language tutoring could be used to scaffold the acquisition and development of this tacit knowledge of programming in novices. The main hypothesis was that natural language tutoring would be more effective at teaching novices how to decompose and plan programs than simply reading the same content.

Given a problem to solve, CPP prescribes the use of open-ended design questions to elicit programming goals (the "what") and the schemata involved in achieving those goals (the "how") from a student. A host of tactics can be used to elicit answers when the student is not able to answer correctly the first time. Many of those identified take advantage of the students' commonsense understandings of problem solving and hand calculations (simulations of the target task). Some tactics are designed to elevate an overly specific answer to something more abstract while others can be used for eliciting important program details, such as identification of all components of a schema. The artifact being designed, and target of much of content of the dialogue, is a natural-language-style pseudocode program. It is constructed in stages: each time a goal is identified and schema fleshed out in dialogue, the corresponding pseudocode steps are integrated into the solution. Students therefore learn to plan in staged fashion and view programs by the schemata that constitute them. Once the pseudocode is complete, the student then pursues an independent implementation phase as they normally would.

ProPl, an automated version of CPP, was also presented. This system conducts natural language dialogue with the student to elicit goals and schemata for a given problem. The role of ProPl is identical to that of CPP: students can use the system to help them prepare to write the program on their own. Dialogue is implemented as Knowledge Construction Dialogues, which walk students through directed lines of reasoning that pose the open-ended questions and execute the tutoring tactics identified in CPP. Variety in student answers is handled by a large semantic grammar as well as by a form-filling approach to handle multiple components in a schema. A check of the resulting dialogues showed that the system did well at choosing the appropriate concept categories when available (84% correct), and that new categories would have been useful about 18% of the time. To act as the control system in an evaluation, a non-dialogue version of ProPl was created that uses canned mini-lessons ("click-through" reading) to present the same tutorial content. Both use the same interface and present identical solutions.

The evaluation was designed to demonstrate the efficacy of natural language dialogue to teach decomposition and composition skills. ProPl, with its use of dialogue, was pitted against the read-only control system. Students in the experiment were randomly assigned to one of the two conditions and used the corresponding system on three programming problems. All participants took a pretest to gauge their incoming programming ability and a written posttest that focused on problem solving ability. In addition, a timed programming posttest was given in the lab, which was followed by a final survey intended to assess their views of the tutoring and feelings towards the system they used. Moreover, copies of all programs submitted to the compiler were collected for many of the students. To evaluate the impact of the two systems, intention-based scoring was used to reveal more accurately how well students were able to assemble algorithms (i.e., solve the composition problem) on their first attempt at each program goal.

In general, the dialogue-based tutoring of ProPl seemed to provide a more meaningful learning experience than the read-only control system. Several outcomes of the experiment support this conclusion:

- Students who used ProPl were consistently better at solving the composition problem than control or baseline students (students who received no tutoring at all). This occurred on the tutored programs and on the written posttest. More specifically, ProPl students generally exhibited fewer errors caused by the interaction of multiple schemata in one program.

- ProPl students also seemed to adopt a more expert-like view of programming by writing code more at the schema level rather than line-by-line, a common trait in novice programmers. Evidence of this was found on the written posttest as well as on certain aspects of the decomposed intention-based scores.

- On the written posttest, students in both conditions demonstrated equivalent competence at solving open-ended decomposition problems. Thus, this experiment provided no evidence that dialogue-based tutoring accelerated students' ability to decompose problems at an abstract level and subsequently express those abstractions in words (the latter is a highly difficult task for novice programmers anyway).

- On the followup survey, control students expressed a preference for the pseudocode over the English descriptions presented in the system; however, ProPl students expressed the reverse view, indicating they found the more abstract representation more appealing.

- Control students claimed to have a better understanding of the material they learned than did ProPl students. Dialogue therefore seemed to have a mitigating effect on the "illusion of knowing" that students often experience.

In sum, ProPl students seemed to develop a stronger understanding of the tacit knowledge of programming and also how to apply it. In some places the differences were not dramatic, and so it certainly cannot be claimed that ProPl students were brought to expert-level. However, it does seem clear that dialogue-based interaction accelerated the acquisition and application of this knowledge.

8.2 CONTRIBUTIONS

This research makes contributions to several streams, including intelligent tutoring systems, computer science education, and dialogue-based educational systems. In addition, teachers of beginning programmers may find the pedagogical presentation useful, perhaps for Teaching Assistant training or for integration of new tactics into their own teaching that are grounded in contemporary theories of cognition and learning.

Contribution #1: An easily incorporated system for teaching programming.

Most systems and pedagogical approaches for novice programmers necessitate significant changes in a course's curriculum. For example, using a new programming language specifically designed for novices requires a change of textbook, compiler, assignments, lecture notes, and so on. The only requirement of ProPl on an introductory course is that it cover structured programming (more about this limitation in the future work section below). To integrate ProPl into a traditional introductory programming course, no changes in programming language, compiler, syllabus, or much of anything else are required. The only real change required is to have students use the system before they begin implementing a solution to an assignment. Since ProPl runs in any Java-enabled browser, this should be a simple matter. Traditional methods of distributing an assignment, like a handout or web page, are passive – only the students who naturally self-explain will do so. ProPl provides a more interactive introduction to problems. It promotes the early confrontation of difficult programming choices that many novices prefer to delay (usually until implementation time). Given the encouraging experimental results, ProPl offers a meaningful pedagogical addition to a programming course with relatively low cost in terms of curricular change.

There is a cost in terms of authoring effort, however (see section 6.3.1.1). It takes roughly 60 hours of authoring and testing time to produce a moderately-sized problem (like Hailstone) to give it sufficient breadth of understanding and enough tutoring tactics to be useful. In addition, the problem statement, pseudocode, and design notes need to be written, but this is, in sum, no more than a 2 hour task (all in HTML). Although a burden in time, the payoff is that KCD authoring skills can be acquired quickly (in my experience) and once problems are written, they can re-used indefinitely. A mundane, but very practical piece of future work would therefore be to build up a library of ProPl problems for use in an introductory curriculum.

Contribution #2: Non-reification of schema/plan knowledge.

In many circles, it is generally assumed that Soloway-style programming plan knowledge should be reified; that is, plans should be explicitly identified, named, and discussed. The work in this dissertation challenges this assumption. One drawback of explicitly teaching plans is that students have the potential to view it as excessive: they already have a lot on the table to learn, including language constructs and how to use the editor/compiler. CPP promotes the learning of plan-based programming knowledge through staged design and the application of the 3-step pattern to elicit goals, schemata, plans, and plan components from students. In a way, the goal is to teach the tacit knowledge of programming in such a way that the student does not realize it is happening. The hope here is that the student will be less likely to reject it as extraneous since it is more closely connected to commonsense and their own language.

One effective way to do this, as this dissertation has laid out, is through natural language tutoring. A possible explanation behind why it worked is that the tutoring strategies help make the connections between novices' intuitive understandings of tasks and the algorithms that implement those tasks more apparent. In other words, the claim made is that it is not necessary to overtly reify this knowledge (i.e., by explicitly teaching schemata in their abstract forms), but rather help the student self-construct through the use of accessible and intuitive notions. Of course, no claim is made regarding the relative strengths of reifying plan knowledge versus not doing it – a different kind of study would be necessary to answer that question. However, the conclusion to take away from this study is that dialogue-based tutoring is a viable alternative to overtly identifying plans in the mutual effort to accelerate students' acquisition of the tacit knowledge of programming.

Contribution #3: Coached Program Planning

One of the aims of this research was to provide an account of tutoring of novice programmers during the planning phase of programming. CPP is such an account and, in its current state, is intended to help students identify programming goals, describe how to achieve those goals, and to actually do so in pseudocode. Along with the staged-design inspired three-step pattern, CPP also includes a library of tutoring tactics implemented as KCDs in ProPl. These tactics are all intended to elicit correct answers from students and help them connect their commonsense understandings to the underlying notions of programming. At least one professor in the U.S. uses the CPP corpus to help train Teaching Assistants, and so the potential value of the CPP model extends beyond its role in the implementation of ProPl.

Contribution #4: Intention-based scoring.

The question of how to evaluate process is a very difficult problem. In programming, online protocols provide a means to do so, but unfortunately, no standard approaches have been developed to judge or score them. In the ProPl experiment, intention-based scoring was proposed as a method to turn programming bugs into comparable scores between students. The approach blends the seminal bug identification techniques of Johnson, Spohrer, and Soloway along with traditional scoring rubrics normally used in programming. The resulting score represents the ability of a student to solve the composition problem on their initial attempts at each programming goal (spread over a protocol).

In addition, because an intention-based score is based on bug categories, it was easy to break the score down into its component parts, thus revealing more about what kinds of problems the students in the two conditions were facing. For example, it was possible to isolate the points lost due to interactions between plans ("merging" errors). To make the comparisons fair, it was also necessary to use the protocols to count the number of goals being achieved to generate the points-lost-per-attempt for each sub-category of bugs. Having done this analysis, it was much easier to interpret the (decomposed) intention-based scores and therefore understand where ProPl was having an impact and where it was not. It is therefore highly likely that this kind of analysis would be applicable to any programming study that is targeting the process of programming and uses online protocols.

8.3 FUTURE WORK

There are many directions to take the work presented in this dissertation. This section discusses possible future work along three dimensions. First, having shown in this dissertation that CPP accelerates the development of the tacit knowledge of programming, it is important to find out why this happened. So, below are some speculation on how to determine this based on analysis of the corpora collected in the experiments. Second, since ProPl is limited in a technological sense, there a number of enhancements that if pursued, might improve the power of the system as a whole. Finally, realizing that CPP (and thus ProPl) only supports structured programming, this section ends with a discussion of what work would be necessary to implement CPP-like support in other programming paradigms.

The ProPl experiment suggested that natural language dialogue was useful with respect to the goal of accelerating learning of the tacit knowledge of programming. Given this finding, the next step is to find out why it worked. In other words, what was it about the dialogues that led to the enhanced skill of ProPl students to solve the composition problem? Given the size of experimental group in the ProPl experiment, it is difficult to cull anything conclusive from the observed dialogue behaviors from it alone. It would be necessary to run the experiment on a larger scale. Some analyses worthy of doing would be to check the KCD success rates, average answer lengths, or perhaps, how close students were to a reasonable answer (when wrong).

In addition, the result that "I don't know" frequency predicts posttest performance suggests that students who struggle with the concepts of programming also struggle to express the ideas of programming in words. The implication is that it may be important to focus on weaker students' abilities to express themselves and draw upon their intuitions first rather than teaching them the details of a particular language. The research question for this line of work would be "does teaching students *how to talk* about programming improve their ability *to plan programs*?"

Turning now to the implementation of ProPl, based on the experiment, the intention-based scoring results suggest that ProPl students produce programs that are "more correct" on their first attempts at achieving program goals. It is not clear whether this initial closeness translated into less frustration and effort overall, however. Deeper analysis is needed to find out whether students in both conditions struggled about the same amount to debug and complete their programs. If true, this suggests that students were possibly unable to recognize a fundamentally correct algorithm unless it compiled and executed as expected, or that they sometimes *undo* correct aspects of their solutions when trying to fix other problems. To address these issues (again, if true), one solution would be to extend tutorial support into the implementation phase. This change could take several forms:

- Integrating ProPl with problem-specific, compile-time feedback, such as that given by PROUST (Johnson 1990), could be used to help novices recall important issues from their tutoring sessions.
- Use ProPl as prescribed in this paper, but provide an environment and tools, such as GPCeditor (Guzdial et al. 1998), to make it easier to apply the ideas learned during the tutoring session.
- Going another step, integration of ProPl-like tutoring into such a novice environment may enhance the learning benefits of such systems.

Indeed, there are a great number of systems for novices (Brusilovsky 1995; Deek 1998) could potentially benefit from the use of natural language tutoring. Yet another option would be to increase the tracking of novices during their independent programming time to capture planning activities and monitor the help they receive. This angle on future research would allow the focus to remain on pre-practice tutoring.

The manipulation check presented in section 7.4 suggested that the KCD-driven dialogues were generally successful with respect to Atlas' classification of answers the authoring of the answer categories. As usual, however, students did learn that short answers were preferred by the system. Also, the output of the KCDs is all canned text, and so it is sometimes difficult for references and anaphora to sound natural. Unless the context is plainly obvious, it is also very difficult to understand when the student uses such language. To take ProPl to the next technological level, one option would be to take more direct advantage of the analysis of tutoring tactics presented in section 4.4 that described them as consisting of a purpose, form, and content. Rather than hard-coding such information directly into the KCDs, it might produce more natural-sounding dialogues if these dimensions were reasoned about at different levels of abstraction (perhaps in a way similar to the 3-tier architecture of BEETLE, Zinn et al. 2003). There are certainly strong connections between the classification of a student's answer and the ensuing tutorial purpose (e.g., an overly specific answer portends an elevation tactic in goal-eliciting contexts). There are undoubtedly countless other possibilities for extending (or replacing) the dialogue manager of ProPl.

Finally, regarding programming paradigm, although the approach presented in this dissertation was directed at structured programming, the technology and general pedagogy underlying ProPl is very much independent of this paradigm. The notions of setting goals and working to achieve them is central to programming in general, and it is likely that the dialogue patterns and tutoring strategies similar to the ones presented in this paper could be applied in other contexts, like object-oriented programming. Even if this did not turn out to be true, the differences would be interesting and might even shed some light on the never-ending discussion over what paradigm is best for beginners. Structured programming was convenient for us given the coverage of the CS0 courses at the University of Pittsburgh and because of the large body of goal/plan research from the mid and late-80's that offered a solid foundation from which to build a tutoring system. As evidenced by intention-based scoring, it also provided an important basis for evaluation.

8.4 CONCLUDING REMARKS

This work was motivated by real students in real classrooms who were really struggling with programming. The original aim was to provide a tool for them that *obliged* them to plan. The reasoning was that since students obviously did not plan on their own, they might encouraged to do so if *asked the right questions.* Thanks to Sacks, Scheglhoff, and Jefferson (1974) and others (Traum and Allen 1994; Graesser, Person, and Magliano 1995), it is known that when one is asked a question, one should respond.[1] Also, having students type answers in natural language rather than selecting answers from a list taps recall memory instead of

[1] Also thanks to our mothers.

recognition memory. For these reasons, CPP was conceived and ProPl was built to get students involved in the planning of their programs, and to help them to be more productive as beginning programmers.

The original goal was to *get students to plan on their own.* Sadly, on the post-test charette (the Count/Hold problem), only 1 student out of 25 sat down with a blank sheet of paper to plan out the program ahead of time. Obviously, changing the habits of novices were beyond the scope of ProPl, but this is, at least for experienced programming instructors, not a surprise at all. On the other hand, the intention-based results and written post-test do suggest that ProPl students learned to assemble steps more productively on their first attempts. It was also found that these same students indicated that the more abstract representations of programming were appealing and useful, whereas the students who read the same material tended to prefer the more concrete pseudocode. These findings are a hint that natural language might be one of the keys to accelerating the acquisition of the elusive tacit knowledge of programming, and provide general support for the notion of pre-practice tutoring.

APPENDIX A

PROGRAMMING PROBLEMS

The *Hailstone problem*, was discussed in detail in section 4.2, page 47. Three other problems were used in this study, and their problem statements appear below. They include a slightly enhanced version of the classic game *Rock-Paper-Scissors* (RPS), a pattern-matching program called *Snap*, and the post-test programming problem, a simple dice game called *Count/Hold* (CH). Students were tutored on the first 3, but not on CH.

A.1 ROCK-PAPER-SCISSORS

Remember the ultimate tie-breaking game of games? Some might think thumb-wrestling or even Monopoly... but none can compare to a good old fashioned game of rock-paper-scissors (RPS). The game is simple: each player pounds their fists into their other hands three times, and on the third, each player comes up with either paper (an open hand), scissors (a v-shape with two fingers), or a rock (fist) . The winner is determined like this:

- paper covers rock
- rock crushes scissors
- scissors cut paper

Many times people like to play best of 3 or 5, or some other number.

The word *match* refers to an entire collection of games played, whereas *game* means one of the many (where each player makes a choice). Thus, the goal of a player is to win the whole match, but it is entirely possible (and likely) that s/he will lose a game or more.

You are to write a program that plays RPS match against the user. The program should ask the user how many times s/he would like to play, then run just enough games to determine a winner. The user will specify the maximum number of games, not the number of games needed to determine a winner.

Your program should determine the computer's choice randomly, and keep the user updated on the progress of the games, including:

- the computer's choice for that game
- the winner of the game
- the current record

When the match is over, your program should indicate who won the match.

A.2 SNAP

Recognizing patterns is a common task for computers. For example, software that monitors cell phone chatter for threatening content can save lives (e.g., preventing terrorism, drug trafficking, etc.).

The basic idea is that there is some stream of information that needs to be monitored. This is costly for humans to do but not for computers. When certain patterns are identified, these monitoring programs are programmed to react in certain ways. For example, it might send a message to a law enforcement officer to watch a particular video feed at the airport.

The pattern in question for this project is the *repetition of numbers*. In other words, you are looking for some number of back-to-back numbers that are all the same. The program should begin by asking the user for how many numbers in a row s/he wants to detect. After this, the program should continually read in (positive) integers until that many numbers that are identical have been entered.

When that number of identical numbers appear in a row, the sequence is "snapped" (hence the name of the project). In other words, the program stops reading in numbers because the proper pattern has been identified. Once this has happened, your program should print out the *average* of all numbers seen up to (but not including) any of the numbers in the desired pattern.

A.3 COUNT/HOLD

There are a lot of fun games that can be played with dice. In this problem, you are being asked to write a program that plays a game with the user. Each player takes turn rolling a die until the game is over.

Each player rolls a 6 sided die privately, covering it up with their hand. Before revealing the result, each player must decide to count the value now, or hold it over:

- If you count the roll, that is your score for the game.
- If you hold it, you have a 0 score for that roll, but you get to add the amount to your next game score. You do not reveal your roll when you hold.

After both players have decided what to do, they reveal their count/hold decision. The winner of the game is the person with the highest score for the roll.

A match is some fixed number of games decided by the players (no no best of calculation). For example, a 10 game match consists of 10 games no matter what. If both players have won the same number of games at the end, the player with the highest total over all of their rolls wins the match. If the tie remains, the whole match ends in a tie.

Here's what to do: write a program that plays a match according to the rules above. Also:

- Do not worry about printing out the rules for the user.
- Let the user pick how long of a match to play.
- Print the match winner when it is all over.

Hint: you do not need arrays for this program.

APPENDIX B

WRITTEN TESTS

The written pre- and post-tests were discussed in section 7.2.3.2. The complete exams are shown below, although they are compressed to preserve space.

B.1 PRETEST

This test is designed to allow us to gauge your general understanding of programming. Please answer the questions to the best of your ability. The results will be used only in conjunction with this study and will be kept confidential.

SECTION 1
Circle the best answer for each question. If you think that more than one answer may be correct, choose the one that seems most appropriate to you.

1. The job of a compiler is to
   a. transform your source code into executable code.
   b. execute your program.
   c. allow you to make changes to your source code and save them.
   d. identify the errors in your program.

2. A byte is consists of 8 bits and a bit is equal to either 0 or 1. How many different values can a byte hold?
   a. 8
   b. 16
   c. 64
   d. 256

3. Which of the following does not conceptually belong?
   a. int
   b. main
   c. char
   d. float

4. In a computer program, which best describes how a variable can get a new value?
   a. the programmer prints the value into the variable
   b. a value is computed and copied into the variables memory location
   c. the computer solves the equation that contains the variable
   d. the compiler assigns the value to the variable

5. Which of the following will produce a syntax error?
   a. accidentally using x instead of * for multiplication
   b. accidentally using + instead of * for multiplication
   c. declaring a variable as a double when you needed an int
   d. inserting too many blank lines between statements

6. To use a variable in your program, the first thing you need to do is
   a. initialize it
   b. declare it
   c. assign it a value
   d. read a value in from the user

7. Which of the following operators has the highest precedence?
   a. `++`
   b. `=`
   c. `+ (binary addition)`
   d. `%`

8. Of the statements below, which does not have the effect of increasing the value of n by 1 upon completion of the statement?
   a. `n++;`
   b. `n = n++;`
   c. `n += 1;`
   d. `n += -n + ++n;`

SECTION 2

For each code segment shown below, determine the final contents of the variables (at the end of code segment) and enter your answers under the variable names on the right.

```
CODE SEGMENTS          FINAL VALUES


int n,m,p;             n m p
n = 45 / 6;
m = 11 % 3;
p = 12 + 3 * 4;
```

```
---------------------------------------------

int a=0, b=2, c=5;    a b c
a = a + 2;
b = b + a;
c = a + b;
---------------------------------------------

int d=100, e=50, f=125; d e f
e += d;
d = (f + d) % 5;
f = e / 100;
---------------------------------------------

int x=1, y=2, z=7;    x y z
z = z * y / 2;
y = x++;
```

SECTION 3
Write code segments that meet the requirements specified below (it is not necessary to write complete programs). Use the programming language you know best. You should declare the variables you need (along with their type).

1. Write a code segment that asks the user for two numbers representing weight (in pounds) and prints out the combined weight (the sum of the two).
2. Write a code segment that reads in two numbers then prints out how many times the first will divide the second and what is left over. For example, if 5 and 13 are entered, the answers are 2 (5 goes into 13 twice) and 3.
3. Write a code segment that begins by reading in a double and accomplishes the following steps:
   a. calculate one-half of the number
   b. calculate one-third of that number
   c. multiply this result by the original number
   d. move the decimal place to the right by two spaces
   e. print the final result

SECTION 4
Determine the output of the following program segment.

```
int hello=5;
System.out.println("Greetings.");
System.out.println("You see, hello starts at " + hello);
System.out.print("but then ");
```

```
if (hello % 2)
hello = 50;

if (hello % 2 == 0)
hello = 100;

System.out.println("it goes to " + hello);
```

B.2 POSTTEST

This test is designed to allow us to gauge your understanding of certain programming concepts. The results will be used only in for this study and will be kept confidential. You have 75 minutes to complete as much as possible.

QUESTION 1

Below you see a bunch of pseudocode programming steps. Your task is to reorganize these steps such that they solve the specified problem.

The problem:

Write a program that repeatedly reads in numbers from the user until a 0 is entered. The program should ignore negative numbers. When a 0 is entered, the program should:

- Print the sum total of all the even numbers
- Print the sum total all the numbers
- Print out how many odd numbers were entered

The steps:

```
print eventotal
print nextval
eventotal = 0
total = total + nextval
eventotal = eventotal + count
if (nextval < 0)
nextval = -1
end while
print count
count = 0
eventotal = eventotal + nextval
increment count
else
while (nextval < 0) do
total = 0
read nextval
while (nextval is not 0) do
if (nextval is even)
while (count > 0) do
if (count > eventotal)
print total
```

Write out your solution here. Remember, just write down the steps from the previous page such that they solve the problem. Things to note:

- Do not add any new steps or change them when you copy them over.
- Not all steps will need to be used.
- Some steps may be used more than once.

QUESTION 2:

This question uses the same problem as before and the same pseudocode steps, but on the following page you will be asked to organize the steps in terms of programming goals.

On this page you can see several programming goals that must be achieved to solve the problem from question 1. Below each write down the steps from the list on the previous page are involved with solving each goal. Again:

- not all of the steps need to be used
- some of the steps need to be used more than once

```
Sum up the even numbers            Count the odd numbers

Read in numbers until seeing 0     Read in a positive number

Sum up all the numbers
```

QUESTION 3:
This has two parts. Just answer the questions as best you can.

Part 1:
Each code segment should be considered independent of the others. Look at each and then try to generally and concisely state what it does. Just try to explain what goal(s) it achieves in a sentence or less. The first one is an example.

```
Code segment:                   Description:

int s=0,c=0,n=1;                Calculate the average of numbers entered
while (n > 0) {                 by the user stopping at 0 (or a negative).
   n = Console.in.readInt();
   s += n;
   c++;
}
System.out.println(s/c);

---------------------------------------------------------------
int n=-1;
while (n<0 || n>100) {
   System.out.print(enter a number: );
   n = Console.in.readInt();
}

---------------------------------------------------------------
int n = Console.in.readInt();
while (n!=4) {
   System.out.println(n);
   if (n%2 == 1)
      n = n*3 + 1;
   else
      n /= 2;
}
```

```
---------------------------------------------------------------
int n,c=0;
while (n >= 0) {
   n = Console.in.readInt();
   if (n >= 0 && n < 10)
     c++;
}
System.out.println(c);

---------------------------------------------------------------
long phn = Console.in.readLong();
while (phn > 1000)
   phn = phn/10;

if (phn==412)
   System.out.println(yes);
```

QUESTION 4:
For this question, consider the following problem statement:

Problem statement:
Write a program that plays a simplified version of War with the user. Your program should deal the user and the computer each a random number between 1 and 10, then print out the winner of that hand. After 21 games, it should print out the overall winner of the match.

What to do:
Pretend that you are a tutor and the student has written the following program. The student is not sure what to do next.

After you read the problem statement, suggest the next two or three things for this student to work on. Try to give the student programming goals related to this problem to pursue, but dont give away too much about how to actually do them.

The program:

```
int user, comp, games=0;

while (games < 21) {
   user = (int) Math.random() * 10 + 1;
   comp = (int) Math.random() * 10 + 1;

   System.out.println(You got a:  + user);
   System.out.println(The computer got:  + comp);

   if (user > comp)
      System.out.println(You won!);
   else if (user < comp)
      System.out.println(The computer won!);
   else
      System.out.println(The game was a tie.);
}
```

QUESTION 5:
You read about the Snap problem earlier during this study. Re-read that problem statement below and then read the directions for this question below that.

The problem:
Write a program that asks the user for how many numbers in a row to detect, then continually read in positive integers until that many identical numbers in a row have been entered. When the specified number of repeats appears in a row, the sequence is snapped and the program should stop reading in numbers. The last thing the program should do is print out the average of all numbers seen, excluding those that caused the sequence to end (those repeated at the end).

For example, if the user wants 4 in a row, and you get: 5 5 2 8 7 7 7 7 The program should stop at the fourth 7, then print 4.0, which is (5+5+2+8)/4

What to do:
Describe how you would go about solving this problem. Just say what you would do and how you would plan to do it in order to solve this problem. Do this at whatever level of detail you want and in any form you want. This is a totally open-ended question.

QUESTION 6:
When you used the software in this study it attempted to help you break a big problem into a series of smaller ones by identifying goals and jotting down ideas to help solve them.

For this last question, read the problem statement below and then read the instructions that come after that.

The Problem:
The game of craps is played by rolling a pair of dice. Depending on the value that comes up on the dice, the game might be over in one roll or it might take several rolls. Here are the rules for one game:

> The player rolls the pair of dice. If the total on the dice is 2, 3, or 12, then the game is lost immediately. If the total on the dice is 7 or 11, then the game is won immediately. If the total is any other number, then the game continues. The total on the dice becomes the "point." The player then rolls the pair of dice over and over until one of two things happen: If the player's roll is a 7, the player loses the game. If the player's roll is equal to the "point" (the total from the first roll), then the player loses. (Otherwise, the player keeps rolling.)

The program that you write should work as follows: The program should first ask the user how many games of craps to play. The program should read the user's input. It should then simulate the specified number of games and should count the number of games that are won.

After all the games have been played, the program should print the number of games that were won and the percentage of games that were won.

What to do:
Write out a list of programming goals and any comments you have about them, similar to the Notes Window you saw while using the software. Just do your best.

APPENDIX C

BUG FREQUENCY RESULTS

As discussed in section 7.3.2.4, there is potential bias in any scoring rubric. In the IBS results presented in this dissertation, points were assigned to various plans and plan components in such a way to emphasize more important aspects of the programming tasks involved. The primary benefit of doing this is that focal aspects of plans can play a proportionally larger role in the score. The drawback, of course, is the above mentioned potential for bias. In this appendix, the raw frequencies of bugs are analyzed and presented in exactly the same manner as in section 7.3.2.3. Since this does not involve a rubric, these results are intended to stand in support of the rubric-derived scores. Only the results are presented in this appendix, as a discussion of these results was provided in sections 7.3.2.4 and 7.3.2.5.

Figure C1 shows the frequencies of merging related errors over the three programming problems. For the Hailstone problem, the control group ($M = .14$, $SD = .20$) produced significantly fewer merging related errors than the baseline group ($M = .68$, $SD = .51$), $t(15) = 2.79$, $p = 0.014$, $es = .96$. The PROPL group ($M = .19$, $SD = .24$) performed similarly well when compared to the baseline group ($t(16) = 2.54$, $p = .022$, $es = .96$). On RPS, the PROPL group ($M = .07$, $SD = .07$) outperformed the baseline group ($M = .21$, $SD = .17$) to a significant level ($t(15) = 2.22$, $p = .042$, $es = .82$). In this case, the difference using the rubric was only marginal. The difference between the PROPL group and control group ($M = .19$, $SD = .20$) failed to maintain marginal significance using bug frequencies ($F(1, 15) = 3.29$, $p = .13$), however (which held using the rubric). Finally, on the Count/Hold project, the PROPL group ($M = .03$, $SD = .10$) again surpassed the control

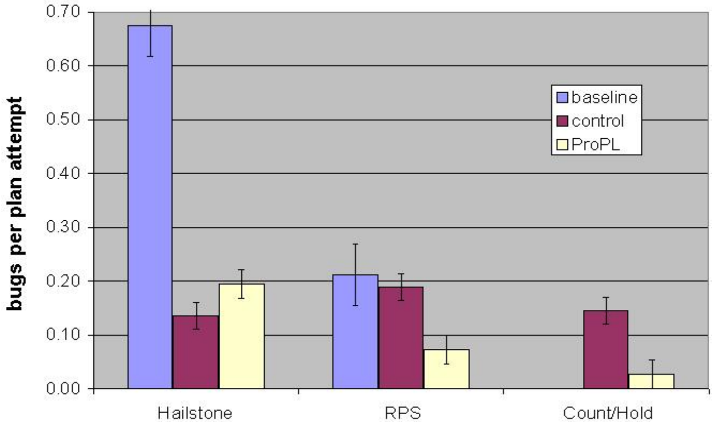


Figure C1: Frequency of merging-related error counts. Standard error bars are shown.

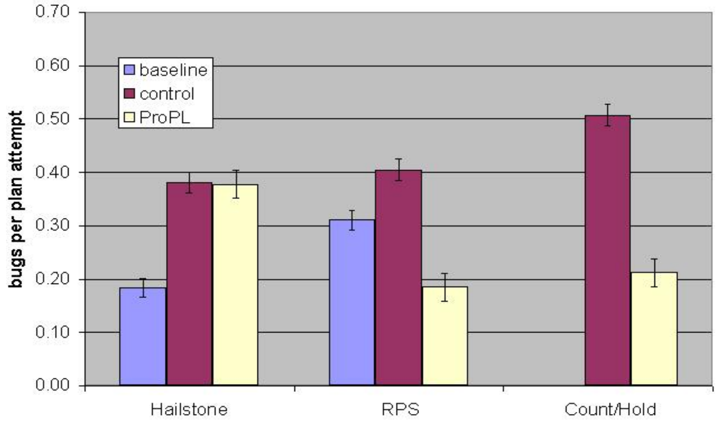


Figure C2: Number of plan component omissions per plan implementation attempt.

group ($M = .15$, $SD = .19$) but this time to a marginally significant level ($F(1, 15) = 3.29$, $p = .072$, $es = .63$).

Next, looking at students' ability to produce complete plans (figure C2), several differences were found to be significant. For the Hailstone problem, the baseline group omitted fewer

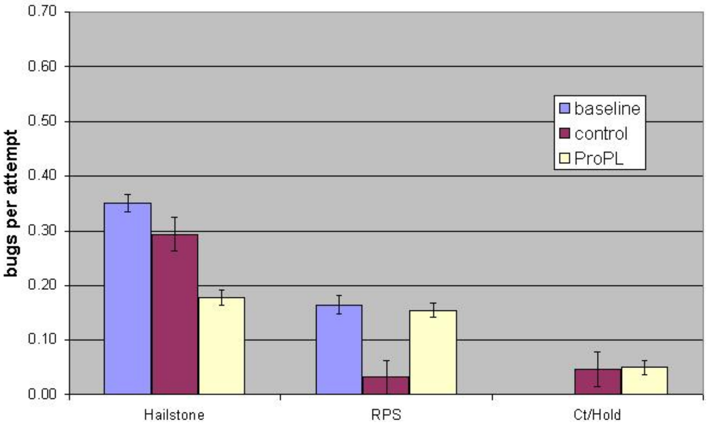


Figure C3: Number of isolated (non-merging) errors committed per plan implementation attempt.

plan parts ($M = .18$, $SD = .16$) than the control group ($M = .38$, $SD = .16$), and to a significant level this time ($t(16) = -2.43$, $p = .029$, $es = 1.25$). When compared to the ProPl group ($M = .38$, $SD = .23$), the difference was found to be marginally significant ($t(15) = -1.97$, $p = .068$, $es = 1.25$ – this difference was not present using the rubric). In the RPS problem, the difference between baseline group ($M = .31$, $SD = .12$) and control group ($M = .40$, $SD = .09$), was not found to be significant using frequencies ($t(16) = -1.65$, $p = .12$), even though it was using the rubric. The ProPl group ($M = .19$, $SD = .15$) was significantly better than the baseline group on RPS, however ($F(1, 15) = 36.8$, $p = .004$, $es = 2.3$). A similar difference also appeared on the posttest project (Count/Hold) with the ProPl group ($M = .21$, $SD = .18$) committing significantly fewer omissions than the control group ($M = .51$, $SD = .31$), $F(1, 15) = 17.7$, $p = .010$, $es = .97$.

Lastly, turning to isolated errors, there were again very few differences between the groups (see figure C3). As before, the only statistically significant difference occurred in the Hailstone problem between the ProPl group ($M = .18$, $SD = .12$) and the baseline group ($M = .35$, $SD = .15$), $t(16) = 2.73$, $p = .015$, $es = 1.1$.

APPENDIX D

SAMPLE KCDS

The files holding the ProPl KCDs covering the three problems in the study take up roughly 150 kilobytes in size (about 75 printed pages of text). In this appendix, a selection of these KCDs are shown, with annotation, to give the reader an idea of the make-up of these knowledge sources. The code below is processed by the KCD tools present in Atlas, compiled in to planning operators, and executed by the APE planner. Note that the expected answer lists in KCDs typically show only a few of the complete list. Synonyms and other answers in the same class were added with the help of the KCD-NLU tool. In many cases, I included a beginning synonym list in comments while writing the KCDs.

D.1 KCD 1: HAND CALCULATION

One of the reasons KCDs were a good fit for implementing many of the tactics of CPP was that interactive examples were easy to implement. Below is a Hailstone KCD that implements one of these introductory examples. The top-level system goal is listed at the top ("`HAILBEGIN`") while the subordinate goals each are a node in the line of reasoning. Other notes:

- Subordinate nodes are named arbitrarily (e.g., HB1, HB2, and so on).
- Immediately after these labels, the system utterance appears in quotes.
- Below this is the expected answer list.

- Correct responses appear alone, while flawed answers sit next to the name of another KCD, which is "called" to remediate for that answer class (these are in all capitals).
- `$anything-else$` is the default category followed when no match is made with expected answers.

This is the simplest kind of KCDs used in ProPl since it is primarily a linear line of reasoning (there is only one path a Hailstone series can take once you know the initial value).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; HAILBEGIN presents an interactive example to the student.  It asks
;; the student to apply the update rules, identify the ground state,
;; determine the count, and identify the largest value.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(goal HAILBEGIN
(goal HB1
"Let's start off with an example.  Please enter just one value at a time.")
(goal HB2
"If we start at 12, what comes next?"
(((("6")   ;"six" is also acceptable (and so on for all #s below)
("37" EVENREC12)
("$anything else$" NEXTVAL12)))))
(goal HB3
"What comes after 6?"
(((("3")
("19" EVENREC6)
("$anything else$" NEXTVAL6)))))
(goal HB4
"Ok, what follows 3?"
(((("10")
("9" FORGOTADDONE)
("1 5" ODDREC3)  ;"one half", took half of 3
("end" WRONGTERMCOND3)  ; student thinks 3 is in the ground state
("$anything else$" NEXTVAL3)))))
(goal HB5
"after 10?"
(((("5")
("end" WRONGTERMCOND10)
("$anything else$" NEXTVAL10)))))
(goal HB6
"good.  next one after 5?"
(((("16")
("15" FORGOTADDONE)
("2 5" ODDREC5) ; "two half"
("end" WRONGTERMCOND5)
("$anything else$" NEXTVAL5)))))
...
```

D.2 KCD 2: ELICITING A GOAL

Two KCDs are shown below. The first elicits generate-sequence, a goal in the Hailstone problem and the second is the remediation tactic for elevating the student from a low-level concept to the more abstract goal. The expected answers are all categories derived from the corpus. Several other aspects of KCD writing not mentioned above are also present in the example:

- Answer categories can be in different classes, but call the same remedial KCDs (for example, "even" and "odd" below). Each class needs to have its own set of synonyms.
- Two KCDs can be called back-to-back ("count" and "largest" below).
- In the example, the most involved tutoring tactic is associated with the *anything-else* branch.

The KCDs mentioned below that begin with "ELICITREPETITION..." are in response to student suggestions to look at another value. This is indicated by mention of words that are related to that notion (e.g., "odd" suggests they are thinking about checking the next number against the rules). In this case, the goal is to elicit the idea of repetition (or looping) from the fact that "getting another value" has been suggested.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ELICITSEQUENCEGOAL - Goal to generate a full sequence
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(goal ELICITSEQUENCEGOAL
(goal ESG1
"Ok, great.  What should we work on now?"
(((("sequence") ("generate more")
("loop" ELICITREPFROMLOOP)
("count" PREMATURECOUNT REASKFORSEQUENCEGOAL)
("largest" PREMATURELARGEST REASKFORSEQUENCEGOAL)
("odd" ELICITREPETITION) ("even" ELICITREPETITION) ("if" ELICITREPETITION)
("ground" ELICITREPFROMTERM) ("end" ELICITREPFROMTERM)
("$anything else$" ELICITSEQUENCEWITHPC)))))

) ; end top-level goal

(goal ELICITREPFROMLOOP
(goal ESG0.1
"That's right.  We will have to add a loop.")
(goal ESG0.2
"Let's figure out why.  What made you think of using a loop?"
(((("sequence") ("many numbers")
("$anything else$" GIVEABSTRACTGOAL))))))
...
```

D.3 KCD 3: FILTERING AND FORM-FILLING

Determining whether or not to engage in an advanced tutoring tactic is not always obvious. I call the solution I came up with *filtering*. This approach, which uses the existing KCD framework, is simply to check the student's answer to the top-level question by searching for known important and relevant keywords. If any are present, then the long and involved example-based tactic is not followed (this is DOLARGESTEXAMPLE below). The drawback is that since there is no memory in KCDs, is that any "goodness" in that initial answer is lost and must be given again when the appropriate questions are asked again later.

This KCD walks the student through the parts of the schema for finding the largest value in a sequence. It is necessary to discuss the initialization, if-statement in the loop, and the print statement. The if-statement in the loop involves several parts: the use of "if", the comparison, and its location. To handle this, the *answer parts* facility provided by KCDs is used (under goal ELS4 below). When asked "how," the system will look for each of these aspects in the answer, and then calls the remaining KCDs to fill in the gaps. This follows the "form-filling" approach to dialogue management. Using answer-parts within KCDs was not originally built for this behavior, but it worked well for ProPl.

```
(goal ELICITLARGESTSCHEMA
(goal ELS1
"Any ideas on how to find the largest?"
(((("greater than") ("compare") ("previous") ;; common parts of good answers
("variable")
("$anything else$" DOLARGESTEXAMPLE)))))
(goal ELS2
"Let's handle the details one at a time.")
(goal ELS3
"We'll need a variable to hold the largest value, let's call it largest.
What should we initialize it to?"
(((("initial") ("hailval")
("0" COULDWORKBUT ELICITINITLARGEST)
("1" COULDWORKBUT ELICITINITLARGEST)
("$anything else$" ELICITINITLARGEST)))))
;; Answer parts provide a nice workaround here...
(goal ELS4
"How are we going to use largest in the program? "
(((("after increment") ("after if") ("before increment") ("before end")
("$anything else$" ELICITLARGECHECKLOC))
  (("assign") ("save")
("$anything else$" ELICITSAVELARGEST))
  (("if")
   ("$anything else$" ELICITIFSTATEMENT))
  (("hailval greater largest") ("largest less hailval")
   ("hailval less largest" COMPARISONREVERSED)
("largest greater hailval" COMPARISONREVERSED)
("compare" COMPARISONGOOD ELICITCOMPARISONDETAILS)
   ("$anything else$" WENEEDTOCOMPARE ELICITCOMPARISONDETAILS))
)))
...
```

BIBLIOGRAPHY

Aleven, Vincent, Ken Koedinger, and Octav Popescu. 2003. “A Tutorial Dialog System to Support Self-Explanation: Evaluation and Open Questions.” Edited by U. Hoppe, F. Verdejo, and J. Kay, *Proceedings of the 11th International Conference on Artificial Intelligence and Education.* Amsterdam: IOS Press, 39–46.

Aleven, Vincent, A. Ogan, Octav Popescu, and Ken Koedinger. 2003. “A Formative Classroom Evaluation of a System that Supports Self-Explanation.” Edited by V. Aleven, U. Hoppe, J. Kay, R. Mizoguchi, H. Pain, and F. Verdejo, *Supplemental Proceedings of the 11th International Conference on Artificial Intelligence (AIED2003), Vol. VI.* School of Information Technologies, University of Sydney, 345–355.

Anderson, John R. 1990. “Analysis of student performance with the LISP tutor.” In *Diagnostic Monitoring of Skill and Knowledge Acquisition*, edited by N. Frederickson, R. Glaser, A. Lesgold, and M. Shaffo, 27–50. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, John R., Albert T. Corbett, Ken R. Koedinger, and R. Pelletier. 1995. “Cognitive tutors: Lessons learned.” *The Journal of the Learning Sciences* 4 (2): 167–207.

Anderson, John R., and B. Reiser. 1985. “The LISP Tutor.” *Byte* 10:1–16.

Ausubel, D. P. 1960. “The use of advance organizers in the learning and retention of meaningful verbal material.” *Journal of Educational Psychology* 53:267–272.

Bloom, B. S. 1984. “The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring.” *Educational Researcher* 13:4–16.

Bonar, Jeffrey G., and Robert Cunningham. 1988. “Bridge: Tutoring the Programming Process.” In *Intelligent Tutoring Systems: Lessons Learned*, edited by Joseph Psotka, L. Dan Massey, and Sharon A. Mutter, 409–434. 1988.

Bonar, Jeffrey G., and Elliot Soloway. 1989. “Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers.” In Soloway and Spohrer 1989, 325–354.

Brown, D. E. 1992. “Using Examples and Analogies to Remediate Misconceptions in Physics: Factors Influencing Conceptual Change.” *Journal of Research in Science Teaching* 29 (1): 17–34.

Brown, J. S., R. R. Burton, and J. Kleer. 1982. "Pedagogical, natural language, and knowledge engineering techniques in SOPHIE I, II, and III." In *Intelligent Tutoring Systems*, edited by D. H. Sleeman and J. S. Brown, 227–282. London: Academic Press.

Bruckman, Amy, and Elizabeth Edwards. 1999. "Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language." *Proceedings of the Conference on Human Factors in Computing Systems.* Pittsburgh, PA, 207–214.

Brusilovsky, Peter. 1995. "Intelligent Learning Environments for Programming." *Proceedings of AI-ED'95, 7th World Conference on Artificial Intelligence in Education.* Washington, DC, 1–8.

Carbonell, J. R. 1970. "AI in CAI: An artificial intelligence approach to computer-assisted instruction." *IEEE Transactions on Man-Machine Systems* 11 (4): 190–202.

Carletta, Jean C. 1996. "Assessing agreement on classification tasks: the Kappa statistic." *Computational Linguistics* 22 (2): 249–254.

Chi, M. T. H., S. Siler, H. Jeong, T. Yamauchi, and R. G. Hausmann. 2001. "Learning from tutoring: A student-centered versus a tutor-centered approach." *Cognitive Science* 25:471–533.

Chi, Micki, P. J. Feltovich, and Robert Glaser. 1981. "Categorization and representation of physics problems by experts and novices." *Cognitive Science* 5:121–152.

Chi, Micki T. H. 1996. "Constructing self-explanations and scaffolded explanations in tutoring." *Applied Cognitive Psychology* 10:S33–S49.

Chi, Micki T. H., M. W. Lewis, P. Reimann, and R. Glaser. 1989. "Self-explanations: How students study and use examples in learning to solve problems." *Cognitive Science* 13 (2): 145–182.

Cohen, Peter A., James A. Kulik, and Chen-Lin C. Kulik. 1982. "Educational outcomes of tutoring: A meta-analysis of findings." *American Educational Research Journal* 19 (2): 237–248.

Collins, Allen, and A. L. Stevens. 1982. "Goals and Strategies of Inquiry Teachers." In *Advances in Instructional Psychology, Vol. 2*, edited by R. Glaser. Hillsdale, NJ: Erlbaum Associates.

Corbett, Albert, and John R. Anderson. 1992. "The LISP intelligent tutoring system: Research in skill acquisition." In *Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration*, edited by J. Larkin, R. Chabay, and C. Scheftic. Hillsdale, NJ: Erlbaum.

———. 1995. “Knowledge decomposition and sugoal reification in the ACT programming tutor.” *Artificial Intelligence and Education: The Proceedings of AI-ED 95*. Charlottesville, VA: AACE.

Core, Mark G., Johanna D. Moore, and Claus Zinn. 2003. “The Role of Initiative in Tutorial Discourse.” *10th Conference of the European Chapter of the Association for Computational Linguistics (to appear)*. Budapest, Hungary.

Deek, Fadi P. 1998. “A Survey and Critical Analysis of Tools for Learning Programming.” *Computer Science Education* 8 (2): 130–178.

———. 1999. “A Framework for an Automated Problem Solving and Program Development Environment.” *Journal of Integrated Design and Process Science* 3 (3): 1–13.

du Boulay, Benedict. 1989. “Some Difficulties of Learning to Program.” In Soloway and Spohrer 1989, 283–299.

Dufresne, Robert J., William J. Gerace, Pamela T. Hardiman, and Jose P. Mestre. 1992. “Constraining Novices to Perform Expertlike Problem Analyses: Effects on Schema Acquisition.” *The Journal of the Learning Sciences* 2 (3): 190–202.

Evens, Martha W., S. Brandle, R. C. Chang, and et. al. 2001. “CIRCSIM-Tutor: An Intelligent Tutoring System Using Natural Language Dialogue.” *Proceedings of the Twelfth Midwest AI and Cognitive Science Conference*. Oxford, OH, 16–23.

Feddon, Jeffrey S., and Neil Charness. 1999. “Component Relationships Depend on Skill in Programming?” *11th Annual PPIG Workshop*. University of Leeds, UK.

Felleisen, Matthias, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press.

Flatt, Matthew. 2002, March. Personal Communication. The Thirty-fourth SIGCSE Technical Symposium on Computer Science Education (SIGCSE), Dr. Scheme Workshop.

Forte, Andrea, and Mark Guzdial. 2004. “Computers for Communication, Not Calculation: Media as a Motivation and Context for Learning.” *Proceedings of 37th Hawaiian International Conference of Systems Sciences*. Big Island, Hawaii.

Freedman, R., Carolyn P. Rose, Michael A. Ringenberg, and Kurt VanLehn. 2000. “ITS Tools for Natural Language Dialogue: A Domain-Independent Parser and Planner.” *Fifth International Conference on Intelligent Tutoring Systems (ITS 2000)*. Springer-Verlag Lecture Notes in Computer Science.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object- Oriented Software*. Addison Wesley.

Gertner, Abigail, and Kurt VanLehn. 2000. “Andes: A Coached Problem Solving Environment for Physics.” Edited by G. Gauthier, C. Frasson, and Kurt VanLehn, *Proceedings of* *the 5th International Conference for Intelligent Tutoring Systems (ITS2000).* Montreal, Canada, 131–142.

Glass, Michael. 2001. "Processing Language Input in the CIRCSIM-Tutor Intelligent Tutoring System." Edited by Johanna Moore, *Artificial Intelligence in Education.* Amsterdam: IOS Press, 210–221.

Glenberg, A. M., A. C. Wilkinson, and W. Epstein. 1982. "The Illusion of Knowing: Failure in the Self-Assessment of Comprehension." *Memory & Cognition* 10:597–602.

Graesser, Arthur C., N. K. Person, and Derrick Harter. 2001. "Teaching Tactics and Dialog in AutoTutor." *International Journal of Artificial Intelligence in Education* 12:257–279.

Graesser, Arthur C., N. K. Person, and J. P. Magliano. 1995. "Collaborative dialogue patterns in naturalistic one-to-one tutoring." *Applied Cognitive Psychology* 9:495–522.

Graesser, Arthur C., Kurt VanLehn, Carolyn P. Ros e, Pamela W. Jordan, and Derek Harter. 2001. "Intelligent Tutoring Systems with Conversational Dialogue." *AI Magazine* 22 (Winter): 39–51.

Graesser, Arthur C., Peter Wiemer-Hastings, N. K. Person, and Derrick Harter. 2000. "Using latent semantic analysis to evaluate the contributions of students in AutoTutor." *Interactive Learning Environments* 8:129–148.

Greeno, J., and H. Simon. 1988. "Problem solving and reasoning." In *Handbook of Experimental Psychology*, edited by R. C. Atkinson, Volume 2, 589–672. New York, NY: John Wiley and Sons.

Guzdial, Mark. 2004, January. "Programming Environments for Novices." In *Computer Science Education Research*, edited by Sally Fincher and Marian Petre. Springer–Verlag.

Guzdial, Mark, Luke Hohmann, Michael Konneman, Christopher Walton, and Elliot Soloway. 1998. "Supporting Programming and Learning-to-Program with an Integrated CAD and Scaffolding Workbench." *Interactive Learning Environments* 6 (1&2): 143–179.

Heffernan, Neil T. 2001, March. "Intelligent Tutoring Systems have Forgotten the Tutor: Adding a Cognitive Model of Human Tutors." Ph.D. diss., Carnegie-Mellon University. Tech Report CMU-CS-01-127.

Herrmann, N., J. C. Popyack, Paul Zoski, C. D. Cera, R. N. Lasas, and A. Nanjappa. 2003. "Redesigning introductory computer programming using multi-level online modules for a mixed audience." *Eighth Annual Innovation and Technology in Computer Science Education (ITiCSE).*

Jadud, Matthew C., and Sally A. Fincher. 2003, March. "Naive tools for studying compilation histories." techreport 3-03, University of Kent Canterbury, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF.

Jeffries, R., A. A. Turner, P. G. Polson, and M. E. Atwood. 1981. “The processes involved in designing software.” In *Cognitive skills and their acquisition*, edited by J. R. Anderson. Hillsdale, NJ: Lawrence Erlbaum Associates.

Johnson, W. Lewis. 1990. “Understanding and Debugging Novice Programs.” *Artificial Intelligence* 42:51–97.

Joni, Saj-Nicole, and Elliot Soloway. 1986. “But My Program Runs! Discourse Rules for Novice Programmers.” *Journal of Educational Computing Research* 2 (1): 95–125.

Jordan, Pamela, Carolyn Rose, and Kurt VanLehn. 2001, May. “Tools for Authoring Tutorial Dialogue Knowledge.” *Proceedings of 11th International Conference on Artificial Intelligence in Education (AIED2001).* Austin, TX.

Jordan, Pamela, and Kurt VanLehn. 2002. “Discourse Processing for Explanatory Essays in Tutorial Applications.” *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialogue.*

Jurafsky, D., and J. H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall.

Kamm, Candace A., Diane J. Litman, and Marilyn A. Walker. 1998, December. “From Novice to Expert: The Effect of Tutorials on User Expertise with Spoken Dialogue Systems.” *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP).* Sydney, Australia, 1211–1214.

Katz, Sandra, David Allbritton, and John Connelly. 2003. “Going Beyond the Problem Given: How Human Tutors Use Post-Solution Discussions to Support Transfer.” *International Journal of Artificial Intelligence in Education* 13:79–116.

Keim, Greg. 1997. “Temperature Based Dialogue in the Duke Programming Tutor.” available on-line at http://citeseer.ist.psu.edu/.

Keim, Greg, M. Fulkerson, and A. Biermann. 1997. Initiative in tutorial dialogue systems.

Lee, P., and C. Phillips. 1998. “Programming Versus Design: Teaching First Year Students.” *SIGCSE Bulliten* 30 (3): 289.

Lepper, M. R., M. Woolverton, D. L. Mumme, and J. L. Gurtner. 1993. “Motivational Techniques of Expert Human Tutors: Lessons for the Design of Computer-Based Tutors.” In *Computers as Cognitive Tools*, 75–105. S. P. LaJoie and S. J. Derry.

Linn, Marcia C. 1985. “The cognitive consequences of programming instruction in classrooms.” *Educational Researcher* 14 (5): 14–16,25–29 (May).

Littman, D., Elliot Soloway, and J. Pinto. 1990. “The Knowledge Required for Tutorial Planning: An Empirical Study.” *Interactive Learning Environments* 1, no. 2.

Mayer, Richard E. 1980. “Elaboration techniques for technical text: An experimental test of the learning strategy hypothesis.” *Journal of Educational Psychology* 72:770–784.

———. 1989. “The Psychology of How Novices Learn Computer Programming.” In Soloway and Spohrer 1989, 129–159.

McArthur, David, Cathleen Stasz, and Mary Zmuidzinas. 1990. “Tutoring Techniques in Algebra.” *Cognition and Instruction* 7 (3): 197–244.

McCalla, Gordon, and K. Murtagh. 1991. “GENIUS: An experiment in ignorance-based automated program advising.” *AISB Quarterly*, pp. 11–19.

McCracken, Michael, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students.” *ACM SIGCSE Bulliten* 33 (4): 125–140. Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-year CS.

McIver, Linda, and Damian Conway. 1999. “GRAIL: A Zeroth Programming Language.” *Proceedings of the International Conference on Computers in Education.*

McKendree, Jean, Bob Radlinski, and Michael E. Atwood. 1992. “The Grace Tutor: A qualified success.” Edited by C. Frasson, G. Gauthier, and G. I. McCalla, *Intelligent Tutoring Systems: Second International Conference, ITS ’92 Proceedings.* Berlin: Springer-Verlag, 677–684.

Merrill, D. C., B. J. Reiser, S. K. Merrill, and S. Landes. 1994. “Guided learning by doing.” *Cognition and Instruction* 13 (3): 315–372.

Merrill, D. C., B. J. Reiser, M. Ranney, and J. G. Trafton. 1992. “Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems.” *The Journal of the Learning Sciences* 2 (3): 277–306.

Miller, L. A. 1981. “Natural language programming: Styles, strategies, and contrasts.” *IBM Systems Journal* 20 (2): 184–215.

Moore, Johanna D. 1995. *Participating in Explanatory Dialogues: Interpreting and Responding to Questions in Context.* Cambridge, MA: ACL-MIT Press.

Newell, Allen, and Herbert Simon. 1972. *Human Problem Solving.* Prentice Hall.

Pane, John F. 2002. “A Programming System for Children that is designed for Usability.” Ph.D. diss., Carnegie Mellong University, Pittsburgh, PA. CMU-CS-02-127.

Pane, John F., Chotirat Ann Ratanamahatana, and Brad Myers. 2001. “Studying the language and structure in non-programmers’ solutions to programming problems.” *International Journal of Human-Computer Studies* 54:237–264.

Paoloa. 2001. “Incorporating Software Visualization in the Design of Intelligent Diagnosis Systems for User Programming.” *Artificial Intelligence Review* 16:61–84.

Pea, Roy D., and Midian D. Kurland. 1983. “On the Cognitive Prequisites of Learning Computer Programming.” Technical Report 18, Bank Street College of Education, New York, NY.

Pennington, Nancy. 1987. “Comprehension strategies in programming.” In *Empirical studies of programmers: second workshop*, edited by Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, 100–113. Norwood, New Jersey: Ablex Corp.

Pennington, Nancy, and B. Grabowski. 1990. “The Tasks of Prorgramming.” In *Psychology of Programming*, edited by J. M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore, 45–62. London: Harcourt Brace Jovanich.

Perkins, D. N., Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1989. “Conditions of Learning in Novice Programmers.” In Soloway and Spohrer 1989, 261–279.

Perkins, D. N., and Fay Martin. 1986. “Fragile Knowledge and Neglected Strategies in Novice Programmers.” In Soloway and Iyenger 1986, 213–220.

Perkins, D. N., Steve Schwartz, and Rebecca Simmons. 1988. “Instructional Strategies for the Problems of Novice Programmers.” In *Teaching and Learning Computer Programming*, edited by Richard E. Mayer, 153–178. Hillsdale, NJ: Lawrence Erlbaum Associates.

Petre, Marian. 1990. “Expert Programmers and Programming Languages.” In *Psychology of Programming*, edited by J. M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore. London: Academic Press.

Pintrich, Paul R., Carl F. Berger, and Paul M. Stemmer. 1987. “Students’ Programming Behavior in a Pascal Course.” *Journal of Research in Science Teaching* 24 (5): 451–466.

Polya, George. 1957. *How to Solve It.* New York: Doubleday–Anchor.

Radlinski, Bob, and Jean McKendree. 1992, May. “Grace meets the “real world”: tutoring COBOL as a second language.” *Proceedings of the SIGCHI conference on Human factors in computing systems.* Monterey, CA: ACM Press, 343–350.

Ramadhan, Haider A., and Benedict Du Boulay. 1993. “Programming Environments for Novices.” In *Cognitive Models and Intelligent Environments for Learning Programming*, edited by G. Dettori, B. Du Boulay, and E. Lemut, 125–134. Berlin: Springer Verlag.

Ramadhan, Haider A., Fadi Deek, and Khalil Shihab. 2001. “Incorporating Software Visualization in the Design of Intelligent Diagnosis Systems for User Programming.” *Artificial Intelligence Review* 16:61–84.

Reiser, Brian J., D. Y. Kimberg, M. C. Lovett, and M. Ranney. 1992. "Knowledge representation and explanation in GIL, an intelligent tutor for programming." In *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*, 111–150. J. H. Larkin and R. W. Chabay.

Renkl, Alexander, R. Stark, H. Gruber, and H. Mandl. 1998. "Learning from worked-out examples: The effects of example variability and elicited self-explanations." *Contemporary Educational Psychology* 23:90–108.

Rich, Charles. 1981. "Inspection methods in programming." Technical Report MIT/AI/TR-604, Massachusetts Institute of Technology, Cambridge, MA.

Rist, Robert S. 1989. "Schema Creation in Programming." *Cognitive Science* 13:389–414.

———. 1995. "Program Structure and Design." *Cognitive Science* 19:507–562.

Robertson, Leslie A. 2000. *Simple Program Design.* Cambridge, MA: Course-Technology – Thompson Learning.

Robertson, S. Ian. 2001. *Problem Solving.* Philadelphia, PA: Psychology Press Ltd.

Robins, Anthony, Janet Rountree, and Nathan Rountree. 2003. "Learning and Teaching Programming: A Review and Discussion." *Computer Science Education* 13 (2): 137–172.

Rose, Carolyn P., D. Bhembe, Stephanie Siler, R. Srivastava, and Kurt VanLehn. 2003a. "Exploring the Effectiveness of Knowledge Construction Dialogues." *Proceedings of AI in Education 2003 Conference.* Sydney, Australia: Amsterdam: IOS Press.

———. 2003b. "The Role of Why Questions in Effective Human Tutoring." *Proceedings of AI in Education 2003 Conference.*

Rose, Carolyn P., Pamela Jordan, Michael Ringenberg, Stephanie Siler, Kurt VanLehn, and Anders Weinstein. 2001. "Interactive Conceptual Tutoring in Atlas-Andes." *Proceedings of AI in Education 2001 Conference.*

Sacks, H., E. A. Schegloff, and G. Jefferson. 1974. "A simplest systematics for the organisation of turn-taking for conversation." *Language* 50:696–735.

Shackelford, Russ. 1993. "Why can't smart students solve simple programming problems?" *International Journal of Man-Machine Studies* 38 (6): 985–997.

———. 1998. *Introduction to Computing and Algorithms.* Addison-Wesley.

Singley, M. K. 1990. "The reification of goal structures in a calculus tutor: Effects on problem solving performance." *Interactive Learning Environments* 1:102–123.

Sleeman, D., A. E. Kelly, A. E. Martinak, R. D. Ward, and J. L. Moore. 1989. "Studies of diagnosis and remediation with high school algebra students." *Cognitive Science* 13:551–568.

Soloway, Elliot. 1989. "Learning to Program = Learning to Construct Mechanisms and Explanations." *Communications of the ACM* 29 (9): 850–858 (September).

Soloway, Elliot, and Kate Ehrlich. 1984. "Empirical Studies of Programming Knowledge." *IEEE Transactions on Software and Engineering* SE-10 (5): 595–609 (September).

Soloway, Elliot, Kate Ehrlich, Jeffrey Bonar, and J. Greenspan. 1982. "What do novices know about programming?" In *Directions in Human-Computer Interaction*, edited by Andre Badre and Ben Schneiderman, 87–122. Norwood, New Jersey: Ablex Corp.

Soloway, Elliot, and Sitharama Iyenger, eds. 1986. *Empirical Studies of Programmers.* Norwood, New Jersey: Ablex Corp.

Soloway, Elliot, and James C. Spohrer, eds. 1989. *Studying the Novice Programmer.* Norwood, New Jersey: Ablex Corp.

Soloway, Elliot, James C. Spohrer, and David Littman. 1988. "E Unum Pluribus: Generating Alternative Designs." In *Teaching and Learning Computer Programming*, edited by Richard E. Mayer. 137–152.

Spohrer, James C. 1992. *MARCEL: Simulating the Novice Programmer.* Norwood, New Jersey: Ablex Corp.

Spohrer, James C., and Elliot Soloway. 1985, November 12-15. "Putting It All Together is Hard For Novice Programmers." *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics.* Tucson, Arizona.

Spohrer, James C., Elliot Soloway, and Edgar Pope. 1989. "A Goal/Plan Analysis of Buggy Pascal Programs." In Soloway and Spohrer 1989, 355–399.

Stevens, A., and Alan Collins. 1977. "The Goal Structure of a Socratic Tutor." *Proceedings of the National ACM Conference.* New York, NY: ACM Press.

Sweller, John. 1994. "Cognitive Load Theory, Learning Difficulty, and Instructional Design." *Learning and Instruction* 4:295–312.

Taylor, Josie. 1999. "Analyzing Novices Analyzing Prolog: What stories do novices tell themselves about Prolog?" In *Learning to Build and Comprehend Complext Information Structures: Prolog as a Case Study*, edited by P. Brna, B. duBoulay, and H. Pain. Norwood, New Jersey: Ablex Corp.

Traum, D. R., and James F. Allen. 1994, June. "Discourse Obligations in Dialogue Processing." *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL-94).* 1–8.

VanLehn, Kurt, Pam Jordan, Carolyn Rose, D. Bhembe, M. Boettner, A. Gaydos, M. Makatchev, U. Pappuswamy, M. Ringenberg, A. Roque, S. Siler, R. Srivastava, and R. Wilson. 2002a. “The Architecture of Why2-Atlas: A Coach for Qualitative Physics Essay Writing.” *Proceedings of the 6th International Conference on Intelligent Tutoring Systems (ITS 2002).* Biarritz, France, 158–167.

VanLehn, Kurt, Collin Lynch, Linnwood Taylor, Anders Weinstein, Richard Shelby, Kate Schulze, D. Treacy, and M. Wintersgill. 2002b. “Minimally invasive tutoring of complex physics problem solving.” Edited by S. A. Cerri, G. Gouarderes, and F. Paraguacu, *Proceedings of the 6th International Conference on Intelligent Tutoring Systems (ITS 2002).* Biarritz, France, 367–376.

VanLehn, Kurt, Stephanie Siler, Chas Murray, and W. B. Baggett. 1998. “What makes a tutorial event effective?” *Proceedings of the Twenty-first Annual Conference of the Cognitive Science Society.* Mahwah, NJ: Lawrence Erlbaum Associates, 1084–1089.

VanLehn, Kurt, Stephanie Siler, Chas Murray, Takashi Yamauchi, and W. B. Baggett. 2003. “Why do only some events caus learning during human tutoring?” *Cognition and Learning* 21 (3): 209–249.

Wasik, B. A., and R. E. Slavin. 1993. “Preventing early reading failure with one-to-one tutoring: A review of five programs.” *Reading Research Quarterly* 28:178–200.

Weber, G., and Peter Brusilovsky. 2001. “ELM-ART: An Adaptive Versatile System for Web-based Instruction.” *International Journal of Artificial Intelligence in Education* 12:351–384.

Wender, Karl F., Gerhard Weber, and Gerd Waloszek. 1987. “Psychological considerations for the design of tutorial systems.” *Proceedings of the Third International Conference on Artificial Intelligence and Education.* Pittsburgh, PA.

Wenger, Etienne, ed. 1987. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to Communication of Knowledge.* Los Altos, CA: Morgan Kaufmann.

White, B., T. A. Shimoda, and J. Frederiksen. 1999. “Enabling students to construct theories of collaborative inquiry and reflective learning: Computer support for metacognitive development.” *International Journal of Artificial Intelligence in Education* 10:151–182.

Winslow, Leon E. 1996. “Programming Pedagogy – A Psychological Overview.” *SIGCSE Bulliten* 28:17–22.

Zinn, Claus, Johanna D. Moore, and Mark G. Core. 2002, June. “A 3-tier Planning Architecture for Managing Tutorial Dialogue.” *Intelligent Tutoring Systems, Sixth International Conference (ITS 2002).* Biarritz, France.

Zinn, Claus, Johanna D. Moore, Mark G. Core, Sebastian Varges, and Kaska Porayska-Pomsta. 2003, September. “The BE&E Tutorial Learning Environment (BEETLE).” *Proceedings of the Seventh Workshop on the Semantics and Pragmatics of Dialogue (Dia-Bruck 2003)*. Saarbrucken, Germany.