Thesis Proposal:

Preventive Tutoring in Programming: A Tutoring System for Novice Program Design

H. Chad Lane
Department of Computer Science
University of Pittsburgh

October 13, 2003

Abstract

To address novice deficiencies in strategic programming knowledge, this thesis proposal describes Coached Program Planning (CPP), a tutorial model for structured program design, and subsequently proposes an intelligent tutoring system embodying that model. The aim is to support novices in the transition from an intuitive understanding of the target task to a deeper, more algorithmic view. This is accomplished by eliciting observations and design decisions from the student before they attempt an implementation. In a human-to-human, computer-mediated study, CPP was found to positively impact novice programming behaviors. In addition, the resulting corpus reveals that that novices are generally able to describe *what* needs to be done (goals) and identify aspects of *how* to achieve those goals (schemas). Not surprisingly, the language they use shows heavy dependence on the problem statement and on their largely syntactic repertoire of programming knowledge. Various tutoring tactics are used to improve flawed student answers, including simple moves like pumping, and more elaborate, multi-turn tactics like the use of interactive concrete examples. I have built a prototype intelligent tutoring system ProPl ("pro-PELL") using knowledge construction dialogues (KCDs) as the primary knowledge source. I am proposing to extend this base system by implementing methods that attempt *KCD recovery.* That is, for utterances not successfully mapped to expected answers, the revised system will execute dialogue moves that attempt to re-establish the line of reasoning modeled in the KCD. The motivation for this approach is the need to accommodate novices' range of linguistic expression of program design concepts. I intend to evaluate the system in terms of problem decomposition skills and general programming competence. I hope to make contributions by enhancing the general understanding of novice programmers and by providing methods for more robust tutorial dialogue.

Contents

1 Introduction

I have taught and tutored novice programmers for close to seven years and am genuinely interested in helping to make life easier for them. As a teacher, I find it painful watching students work so diligently, but do things that, in my view, are simply not productive. What makes programming so hard for beginners? Why do novice programmers adopt such poor problem solving and programming strategies? What can be done to avert these chronic behaviors? There are obviously many answers to these questions. To the question of what makes programming hard, Mark Guzdial points out that each answer implies a family of environments based on that answer. He continues:

> Not all of these potential environments have been built and explored, however. The field of Computer Science Education Research is too new, and there are too few people doing work in this field. We are still in the stage of the field of identifying potential answers to key questions–indeed, even figuring out what the key questions are! [22, p. 1]

I seek to build a system for novice programmers that tutors students on how to *begin* writing a program. Rather than tutoring students on how to fix poorly conceived, bug-ridden programs, I want to help students learn to give proper credence to understanding problems and designing solutions.

In this proposal I will describe the pedagogical problem novice programming presents and discuss the results of a study I have completed on *Coached Program Planning* (CPP), a style of tutoring I have developed for novice program design. I will describe the basic elements of CPP and present an analysis of the resulting corpus. In particular, I focus on the basic structure of the tutoring dialogues, the tactics used by the tutor, and the kinds of answers given by students. From this, I move into a description of ProPl ("PROgram PLanner," pronounced "pro-PELL"), the intelligent tutoring system I am building, and finish with a discussion of my proposed extensions to this system.

These proposed extensions are all based on the notion of increasing the ability of ProPl to understand and react to unexpected student contributions. A major goal of this work is therefore to develop a framework in support of more robust understanding of students in tutorial dialogue systems. My research is a result of a great deal of interaction and study of novice programmers, and as such, I hope also to make contributions to our understandings of how they go about the task of programming. There has been much work targeting novices before, during, and after programming, but much less involving the *tutoring* of novices *before* they enter into an implementation phase. What is this tutoring like? What kinds of tutoring strategies are effective in eliciting designs from students? I hope to lay groundwork for many of these issues.

The symbiotic relationship between my teaching and research began around January of 2001 when I started thinking about systems for novice programmers. Insights from years of observing novices struggle with programming have influenced the pedagogical stands I have taken in my research. Likewise, the data I have studied has given me a glimpse into the mind of a novice that few teachers ever have the chance to see. This has changed my views on what I think novices should be taught in classrooms and how best to do that. This continued connection, I feel, has helped me maintain purpose and direction in both lives. I hope to bring this to fruition over the next year as I develop ProPl, use it with students, and observe its impact.

2 Background and Motivation

In this section I will describe the motivating educational problem being addressed by my work and argue for the preventive tutoring of programming.

2.1 Cognitive subtasks involved in programming

The tutorial model I will be promoting has its roots in the model of general problem solving originally proposed by Polya [41], Newell and Simon [34], and then later recast for the task of programming by Pea and Kurland and others [14, 36]. The model posits four necessary stages novices and experts go through when writing a computer program:

1. **understanding the problem** – develop a mental representation of the task
2. **planning a solution** – produce a description (mental or on paper) of the program
3. **implementing a solution** – translate design into program code
4. **comprehension and debugging of the solution** – observe program execution and repair if necessary

This is an idealized model. Experts are able to interweave phases while novices tend to jump immediately to the implementation phase [40]. In any case, I work from the assumption that this is an appropriate model from which to situate the tutoring of programming. My focus will be on the pre-implementation phases of problem understanding and planning.

2.2 Novice Programming

Evaluations of programming skill done in the 80's [48] and again in the late 90's [31] revealed an alarming lack of competence, even after one year of programming instruction. It is clear that programming represents a significant pedagogical challenge.

What makes programming hard?

Novices struggle with just about every aspect of programming [12, 47, 51], and adopt behaviors that frequently worsen the situation for them [12, 40]. Even their incoming knowledge can be a hindrance [4]. Pennington [37] found that a large number of mistakes stemmed from flawed understandings of goal and subgoal structure. In addition, novices exacerbate the difficulties they face by doing what they do. For example, they tend to begin the programming process by keying in code [40, 55]. This tendency is been confirmed by numerous studies showing that novices use a bottom-up approach to programming [44, 46, 55], thinking of programming in terms of small, syntactic units. This is not peculiar to programming: novices in other domains (like physics) display similar behaviors [7].

Novices also struggle a great deal with assembling program parts that accomplish different goals. Spohrer and Soloway [51] found that nearly 80% of novice bugs stemmed from incorrect *plan-merging*.[1] In other words, novices were generally able to find the right plans, but that the interactions between these plans represented too great a challenge.

Another particularly detrimental behavior displayed by many novices is the use of program output behavior as the primary means of judging program quality [26]. This view is substantiated by the observation that many novices resort to "tinkering" as a debugging strategy [38]. That is, such students repeatedly 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. So why don't novices plan? A viable explanation is that novices do not plan simply because they do not know how [12, 13, 40, 39]. 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

[1] "Putting the pieces together."

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. 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.

Programming and design

It is generally accepted that design involves the creation of some form of intermediate representation which bridges a problem statement to an implemented program [36]. A key aspect to most notions of design is *problem decomposition* which, according to Soloway, asks the programmer to "lay the components of the solution on the table, that is, *decompose* the problem and *identify* the solution components" [50, p. 142]. These components include:

- **goals** the objectives as indicated by the problem statement
- **schemas** the methods by which the goals are achieved
- **objects** the data items that will be needed in a solution

The purpose of going through a design phase is therefore to identify these components in one form or another. A schema can be thought of as a stereotypical program fragment that achieves a well-defined goal. Novices, by definition, lack a library of such schemas. One of the aims of this research is to explore this deficiency and develop ways to help novices overcome it.

2.3 Human tutoring

It is generally acknowledged that the most effective form of learning occurs in one-to-one, expert human tutoring [2]. Recent evidence suggests that learning is correlated with an increased student role in the dialogue [8, 9]. Earlier work in human tutoring research focused on the tutor's behavior, assuming that learning was a function of skilled execution of tutoring tactics [20, 30]. Blending these views, the research problem becomes one of finding tutoring tactics that elicit from the student the kinds of deeper and more significant contributions associated with increased learning. A subsidiary goal, and the one I am more directly addressing, is that tutors (automated and human) need to take appropriate steps to *recognize these deeper student contributions* when they are present.

2.4 An argument for preventive tutoring

Given the natural tendencies of novices, it is unlikely that they will ask, or even know to ask, many program design questions until implementation time. Thus, novices are forced into "on-the-fly" planning of their program, something for which they are particularly ill-equipped [36, 38]. By intervening early in the programming process, these issues can be addressed out of the context of other complicating factors (like the programming language, compiler, editor, etc.). In the language of cognitive load theory [53], if the elements to be learned can be taught sequentially rather than simultaneously, then the cognitive load students face will be reduced. What this means for programming is that if the issues of problem understanding and design can be addressed early, then students will have less to deal with at implementation time. Perkins relates this view to problem decomposition skills:

> 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. [38, p. 275]

Clearly, the natural tendencies of novices are not conducive to success in programming.

Given a problem statement, students are *not* likely to have the ability articulate a clear set of programming goals. They are, however, likely to have the ability to suggest some applicable lower level programming concepts (e.g., "we need an if statement"). Indeed, I have confirmed this in my research and am using these findings to drive the design of an intelligent tutoring system. I am proposing that design be taught *in concert* with student projects and their natural, bottom-up tendencies. The basic approach is to *take advantage of natural novice tendencies* and use them to steer novices into to a more expert-like, abstract view. Such an approach can only be done within a preventive tutoring model.

3 Tutoring Novice Program Design

Coached Program Planning (CPP) is a style of tutoring I have developed specifically aimed at addressing the issues raised in the previous section. In this section, I will describe CPP and attempt to characterize the behaviors of both the student and tutor in a small corpus of CPP dialogues. An entire protocol from a CPP tutoring session can be found in appendix C as well the results from a controlled evaluation in appendix D.

3.1 General aims

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. This involves the elicitation of important aspects of both the problem and the desired solution. At the very least, these include identification of programming goals, schemas suitable to meet these goals, and applicable programming primitives (variables, loops, etc.). In CPP, these are realized through the creation of natural-language style pseudocode.

Cultivating an algorithmic mindset

Most computer science educators would agree that the ultimate goal of teaching programming is to help students develop a sense of algorithm. This is obviously a challenging endeavor. As an example, consider the problem of playing the classic game of Rock-Paper-Scissors (RPS). Most people already know how to play, and if not, can be easily taught. The act of playing a game is essentially a simulation of the desired program behavior. I will refer to this as a *hand calculation*, terminology introduced by Spohrer [51]. If a novice is asked to write a program that plays RPS with a person, however, the game suddenly becomes more complicated. There are many issues to resolve: 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.

A goal of CPP is therefore to help novices make a transition to a more algorithmic view of a task and confront important design questions prior to an implementation. We will say that if a student is able to perform a hand calculation of the target task, s/he has an *intuitive understanding* of the problem. An intuitive understanding for RPS corresponds to an ability to play a game. The more elaborate *algorithmic understanding* is established when the student has acquired the pieces of knowledge needed to implement a solution (the ultimate evidence being the final program). As we will discuss in section 3.5, the tutorial use of hand calculations (i.e., concrete examples) are ideal when helping students graduate from an intuitive understanding into an algorithmic one.

What is being learned?

The "pieces of knowledge" referred to in the last section may best be viewed as Soloway-style programming plans [46] (discussed earlier in section 2.2) or patterns [15] . Whether this schematic knowledge should be reified is unclear. Soloway has always supported the explicit teaching of plans and a number of systems have been built that do this, including BRIDGE [3], GPC-Editor [23], and the TELC-tutor [10]. My research challenges this assumption by helping students construct this plan knowledge based largely on intuitive notions of the task and from their largely-syntactic store of programming knowledge. This is discussed further in section 6.

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 1: The Hailstone Problem

Targeted problem types

Introductory programming assignments generally involve *knowledge-lean* tasks. Such tasks require minimal specialized domain knowledge to comprehend and perform by hand. Some assigned tasks are overt, like converting a temperature from Fahrenheit to Celsius. Others, while still trivial to perform, involve assumptions and hidden steps that are easy to overlook. These kinds of problems often require some piece of insight to produce an algorithmic rendition of the target task. As mentioned earlier, Rock-Paper-Scissors is such a task. Another example I will draw from is the Hailstone problem, shown in figure 1. For most novices, writing a program to solve Hailstone is very challenging. Appendix A contains a more complete discussion including a pseudocode solution and a discussion of the goal structure.

1. identify a programming *goal*
2. describe a *schema* for attaining this goal
3. suggest pseudocode *step*s that achieve the goal
4. *place* the steps within the pseudocode

Figure 2: **Four-step pattern**

3.2 Elements of a tutoring session

The collaborative goal of the tutor and student in CPP is to build a natural-language style pseudocode solution. This solution plays the role of a design that the student can then use to guide an independent (untutored) implementation.

Sessions always begin with an interactive hand calculation. This is done to confirm the student's understanding of the target task and to act as a demonstration the desired program behavior. After this, the goal shifts to writing pseudocode. Globally, the tutor's effort is driven by repeated application of a four-step tutoring pattern shown in figure 2.[2] It is repeated until all programming goals have been satisfied, and the pseudocode is complete. It should be noted this does *not* typically include the reification of schemas (this happens with pseudocode), but rather the intuitive description of their parts. Not surprisingly, the four-step pattern heavily influences the structure of the dialogue (discussed further in section 3.3).

An environment for CPP appears in figure 3. The interface consists of three main windows. A mini-browser (upper left) displays HTML pages and remains available throughout the tutoring session. The dialogue window (lower left) allows communication between a human tutor and the student. Finally, the pseudocode window (right half) contains drag-

[2]It is perhaps more accurate to say "four-step prescription," however, it was not evident until I started analyzing the dialogues and observed this structure.

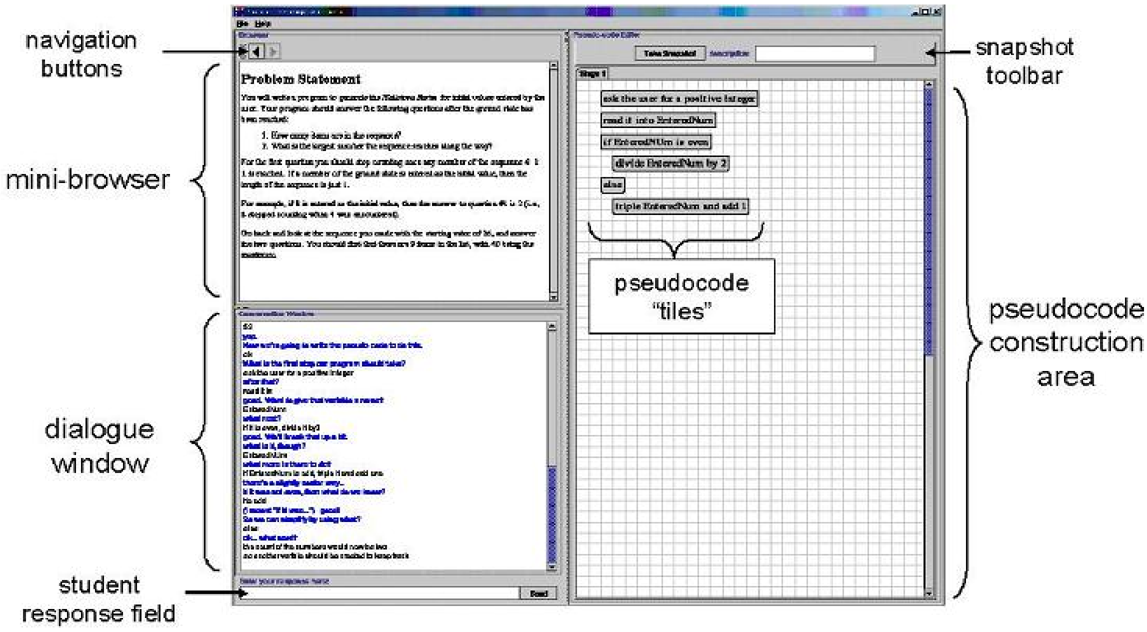

Figure 3: CPP Environment

gable *tiles* that contain text, each of which represents a step in the solution. A tutoring session begins with the student reading the problem statement in the browser, followed by a hand calculation, and then on to pseudocode construction. As the dialogue proceeds, pseudocode steps emerge, the tutor creates tiles, and the student drags them into the pseudocode area. 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. 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.

3.3 Corpus overview

The full CPP Corpus consists of 48 tutoring sessions across five problems, all performed between the summer of 2001 and the summer of 2003.[3] A description of the problems in the corpus can be found in appendix B. My analysis of has focused primarily on Hailstone and to a lesser extent, RPS. Of the 48, 22 are from the Hailstone problem and 12 from RPS. Of these, 6 Hailstone and 3 RPS dialogues have been set aside for initial testing of ProPl. The discussion presented here is based on the 16 opened Hailstone sessions. It should be also be noted that 3 of the Hailstone and 4 four the RPS sessions (done in the summer of 2003) do *not* consist of full CPP. More specifically, pseudocode was dropped in these sessions to assess the feasibility of goal- and schema-only tutoring.

3.4 Analysis of student utterances

This section provides a characterization of student utterances in response to top-level questions asked by the tutor. In terms of the four-step pattern, these refer to how students describe the goals they intend to pursue and *how* they intend to do so (i.e., schemas).

How student talk about programming goals

To elicit a new programming goal, the tutor typically asks something like "What should we work on now?" Taskwise, these are very much *closed-answer* questions, meaning there is only a small number of acceptable answers. However, as we will see, there are many acceptable alternatives for expressing goals. In Hailstone, five distinct goals are pursued by the tutor (see appendix A for a detailed discussion). For each, there is a transition point in the dialogue when the tutor attempts to elicit a new goal from the student. Some examples

[3] I was tutor in this study, but will continue to refer to myself as "the tutor" when speaking of the me in the corpus.

1. **get-initial-value:** "read an integer from a user"
2. **get-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 4: **Example goal suggestions from CPP corpus.** Goals associated with each utterance appear in boldface.

students' responses appear in figure 4.

Table 1 summarizes all of the student answers to such questions from the 16 Hailstone dialogues I have analyzed.[4] 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).[5] There are three main observations to make based on this data:

Observation 1: *Students seem to have little difficulty saying something productive.* The high frequency answer categories were all considered valid ways of expressing the goals for which they are associated, at least in the eyes of the tutor.

[4] All analysis was done solely by me, so these conclusions should be considered preliminary.

[5] 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 mentioned earlier."

goal	*common answer parts*	*frequency*	*avg len*
get-initial-value	read, prompt, pick, integer, number	75% (12/16)	6.5
	odd, even, if	19% (3/16)	
get-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-items	count, how many, items in sequence	88% (14/16)	10.1
find-largest	find, largest, highest, maximum	100% (14/14)	8.0

Table 1: Goal utterance summary in Hailstone

Observation 2: *If the problem statement contains a clearly stated programming goal, students are better able to state it in the dialogue.* This is confirmed by noting that in the problem statement (figure 1, page 12), the clearly stated goals in the problem statement, *count-items* and *find-largest*, are also the ones students had the most success in answering. This observation provides a possible explanation for the apparent success of novices at making good goal suggestions. It is also evidence that students pay attention to the problem statement.

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 1 for the goals not explicitly given (*get-initial-value*, *get-next-value*, and *generate-sequence*), the answer parts for these are not nearly at the level of abstraction of the explicitly stated goals. This behavior suggests that students *talk* about programming in the same way they actually program: bottom-up.

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: Example schema suggestions from CPP corpus

How students talk about "how"

Given the nature of students' responses to goal-eliciting questions, it is no surprise that many of those answers can lead quickly into a discussion of how to achieve the goal. The typical path the tutor takes in these instances 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 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. 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 3.5)

It should be noted that there were no instances of the tutor asking a student how to accomplish either the *get-input* or *get-next-value* goals. This was because the "how" was

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-items	**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)

Table 2: **Schema suggestions in Hailstone.** 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.

already evident based on the answers to the goal questions and in the creation of pseudocode. Thus, my analysis of Hailstone consists of the remaining three goals. I collected all instances of when students were asked schema-eliciting questions. The summary appears in table 2. Instead of organizing by utterances this time, the table organizes by concepts. This is done because a schema typically consists of several parts, and thus the understanding task is to identify these answer parts. The frequency, therefore, shows how often each concept appears in utterances for that goal. Many utterances contained 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 of a student to talk about "how" to achieve the associated goals. At times, two such utterances were included (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 2 confirms a few of the earlier observations made about goals. Although it takes some prodding at times, students still seem to generally 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.

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. Performing this kind of reasoning is at heart of my proposed research discussed in section 5.

Pseudocode

Most of the pseudocode oriented remediation in the corpus has to do with step placement. For example, it was common to see students improperly indent steps, or placing them out of order. Because my focus has been purely on the goal and schema issues, I will not include further discussion of the pseudocode tutoring in this proposal.

1. **content-free pump:** "Can you tell me more about that?"
2. **refer to problem statement:** "You might want to take a look at the problem statement."
3. **elicit requisite observation:** "Does this program generate a sequence?"
4. **hint:** "Think about what allows us to repeat something over and over."
5. **hypothetical example:** "So when you are doing a sequence, how much does the count go up each time you get a new value?"

Figure 6: Simple tutoring tactics

3.5 Tutoring tactics

Expert tutors tend to elicit information from students as opposed to simply providing it [18]. This has the dual effect of maximizing the participation of the student and making him/her more responsible for the problem solving. In CPP, each part of the four step pattern (section 3) generally corresponds to a tutor action (typically a question) and a student response. The topic of this section is to describe what the tutor does when these initial answers are lacking.

Simple tactics

When student answers are not ideal, the tutor engages some tactic intended to elicit some piece of information. These can be goals, simple observations, abstractions, or pseudocode steps. Several simple tactics are shown in figure 6. These are generally one or two turn in length, often ending with the tutor refining, or affirming that the student has provided the intended information.

As discussed in section 3.4, students are generally able to point at goals. More specifically, they tend to say something relevant to a goal the tutor has in mind at the goal-transition

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 7: Elicitation pattern

points during CPP tutoring. However, asking students to describe a schema, or "how" to proceed is much more challenging. Often the goal of describing a programming schema to achieve a goal involves some important abstraction. For example, in Hailstone, describing which two values need to be compared to find the largest is such an abstraction. To elicit these from students, and when their answers to simple strategies are lacking, the most commonly used tactic is to engage a concrete example (i.e., do a hand calculation with the student). The general pattern followed by the tutor is shown in figure 7. This outline subsumes the simple tactics from figure 6 with steps 1 and 4.

The appealing property of examples is that they allow the tutor to ask isolated and simple questions that the student can answer. They are used for at least two reasons: to have the student reflect on his/her own intuitive simulation of the task and to help a student "decompile" this knowledge.

Reflection tactics

Knowledge-lean tasks provide fertile ground for tutoring novice program design. Asking the student to self-report during the process of a hand calculation can bring important issues to the table, and with the tutor's help, act as a bridge into appropriate planning knowledge.

Reflection is used in two distinct ways in the corpus. The first is to set the stage for making a distinction between intuitive and algorithmic views of the problem. By having the student clearly state how to accomplish a goal intuitively, the approach can either be accepted or rejected by the tutor as a way to go about solving it algorithmically. For example, line 4 of figure 8, the student indicates s/he is confused. However, after being asked about how s/he accomplished it intuitively, (lines 5 and 6), the student actually reveals a usable strategy. Most students who were asked reported a "post-sequence" counting strategy (e.g., "I just counted them up."), in which case the tutor would gently reject the idea.

This second way reflection is used is to help reveal the intricacies and hidden mental steps of the task. Not surprisingly, the usual strategy reported for finding the largest value was to simply scan over the values and pick out the largest. To reveal the details involved in finding the largest, an concrete example can be used, asking the student along the way to identify the current largest, updating when necessary. The key abstraction (insight) to be elicited here is to identify which two values are involved the comparison.

0	T	What should we work on next?
1	S	**we need to count how many numbers were in the sequence.**
2	T	right. Any ideas on how to do that?
3	T	what step we should add?
4	S	**not sure...**
5	T	that's ok... in our previous example that started with 12 how did you count that one?
6	S	**i just went up and started with 12 and counted how many numbers were in the series until 4 was reached.**
7	T	good. so when you started with 12 what was the count at?
8	S	**0**
9	T	well... we do count it as part of the series.
10	T	so 1.
11	S	**oh yeah**
12	S	**right**
13	T	and after that was 6. What's the count now?
14	S	**2**
15	T	good. next is 3. count?
16	S	**3**
17	T	good. Do you see the algorithm yet?
18	S	**we are adding 1 to the count**
19	T	good. And how often do we do that?
20	S	**everytime we go through the loop**
21	T	right! One time through the loop corresponds to one new hailstone value.
22	T	That is a crucial observation.

Figure 8: Learning to count algorithmically

Decompiling acquired knowledge

Examples can also be used to help students decompile procedural knowledge that could be useful in an algorithmic context. For example, the dialogue in figure 8 shows the tutor demonstrating how the count can be done *dynamically*. By going step-by-step, and asking simple questions along the way, the student is able to keep up and answer the questions. In the end, the crucial observation is made (line 20), and dialogue involving where the counter goes in relation to the loop can ensue. Relating this to underlying schema-knowledge, the schema for counting a sequence (general notion) includes the presence of an increment *inside* the enclosing loop. This is the observation being sought by the tutor.

Another example of this comes from RPS. The game is so familiar (to most) that playing a game requires minimal thought – it is compiled knowledge. The actions of making a selection, viewing the opponent's selection, comparing the two, deciding the result, and reacting to it are all distinct steps. To help the student decompile in this context, playing an actual game (i.e., the example) and *slowing* down to observe the steps makes this possible, and easily accessible to the student. Decompilation also occurs as part of the reflection tactics mentioned above, however in this case it is not being used to contrast with the intuitive approach.

The effectiveness of concrete examples

Of course, the mere existence of examples in the corpus is not sufficient to justify their repeated use. In general, the use of examples in tutoring is known to be an effective strategy in tutoring [5, 24, 43]. This seems to also hold true in an algorithmic context. Across the 16 Hailstone dialogues that I have analyzed, there was a total of 37 uses of concrete examples, not including the obligatory beginning hand calculations. Of these, 5 were directly related to the placement of pseudocode steps and 5 others were used to elaborate the problem statement to the student. This leaves 27 instances, all of which were intended to elicit some answer from the student. To demonstrate the effectiveness of the example-driven tactics, I looked at the within-dialogue success of each of these subdialogues. In other words, each time the tutor engaged an example, I considered the tactic successful if in the end, the student provided the complete expected answer of the tutor. If the tutor had to give away part or all of the answer in the end, then I labeled the tactic a failure. Of the 27 instances, the student failed to provide the targeted answer only 4 times. The failures seem to 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.

4 ITS Design Considerations

4.1 Brief review of novice programming systems

A good portion of work on intelligent programming environments seems to avoid or minimize issues of design. Many of these systems are language-driven and focus on debugging, like the seminal plan-based bug finder PROUST [25] and the Prolog bug finder [16]. Others are support smaller scale programming (e.g., one function at a time) like the Carnegie-Mellon LISP Tutor [1] and GRACE [32]. GIL [43] is another such system, most notable here due to its heavy reliance on concrete examples which are used to drive the writing of generalized LISP functions. Examples are also the centerpiece to the ITS component of ELM-ART [54], but are retrieved *en masse*, based on a student model and for the purpose of code reuse. By working at the code level, these systems are essentially supporting on-the-fly planning of programs. This works well for small scale assignments and for the learning of particular programming languages. For programming in the slightly larger, however, the tutorial intervention may be too late for many novices. Early misconceptions have the chance to propagate throughout the whole programming process and important design issues (and their consequences) may linger until as late as the debugging phase.

An exception is found with BRIDGE, a system that helps students construct a solution description in natural language, followed by successive rewrites in more precise forms, ultimately resulting in a working program [3]. Although BRIDGE did not skirt design, it was accomplished via menu selections and thus amounted to a task of recognition instead of generation. In addition, BRIDGE reified plans as an intermediate language in the form of puzzle pieces. Thus, to write a program, a student had to create a natural language solution, translate that into puzzle pieces, then translate that into Pascal. It is possible that students viewed these as extra languages to learn. No conclusive pedagogical benefits regarding the use of BRIDGE were ever published.

More recently, a trend towards supporting design is evident. Several experimental systems have emerged that attack, in a direct way, the questions of design and planning for novices. For example, Guzdial's GPC-Editor reifies programming plans and goals [23], but provides no intelligent support. Thus students are free to explore, but are not given feedback (by the system) regarding the correctness of their approach. SolveIt [11] is another similar system that explicitly includes a design phase, but again, does not provide intelligent analysis or support of students' solutions. A lack of intelligence is not in and of itself a problem. However, without a way to judge correctness, the risk of garden paths is unavoidable. Some students, especially the so-called "movers" [39], are especially prone to following them.

Discover [42] 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 given immediate negative feedback when their solution does not follow that prescribed by the model. Pseudocode is constructed with the help of buttons which, when pressed, produce pseudocode templates that the student fills in. Discover therefore draws on recognition memory and requires the learning of a new language (albeit a thoughtfully constructed one).

4.2 Dialogue considerations

Why dialogue?

In order to avoid the drawbacks of the systems discussed in the last section, novice systems should eschew menu-based interfaces in favor of those that require that students provide answers in their entirety. The idea is to tap *recall* memory as opposed to *recognition* memory. The most obvious way to achieve this is to have the student to 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 [1, 32, 43]. By starting with a specific programming language, this approach seems contrary to the aim of helping novices develop a problem-driven, more abstract view of programming.

The approach I am adopting is to have the student provide a description of *how* s/he intends go about solving the problem (i.e., produce a design). The inclusion of menus or selection-oriented features would necessarily trigger recognition memory, and so we are left with either the introduction of a specialized (intermediate) design language or the use of natural language. New languages introduce new learning hurdles, but natural language is already familiar to the student. Students can therefore express concepts in their own words and connections to existing knowledge can be made more readily. The use of natural language *dialogue* allows the tutor to be involved as the student contemplates design issues and attempts to describe these ideas. It is for these reasons I seek to build a tutorial dialogue system.

Understanding student input

Several different approaches to understanding are used in some recent tutorial dialogue systems. Atlas [45] includes CARMEL, a parser implementing a form of a semantic grammar used to match student utterances to expected answers.[6] CIRCSIM [17] uses a large number of specialized finite transducers, all programmed to handle certain kinds of answers. Because only closed-answer questions are ever asked and student answers are generally short, this technique works well in their domain (Cardiovascular physiology). From the numerical camp, AutoTutor [21] employs Latent Semantic Analysis (LSA), a shallow understanding approach, to identify the presence of concepts in student utterances.

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

[6]CARMEL provides more elaborate functionality, but this is not yet used in the dialogue portion of Atlas.

relevant goal from the student's answer. An ability to make this kind of inference will be essential if 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. As we have seen, this often results in a useful, productive answer deserving of tutorial recognition and pursuit. I have argued that in design-oriented tutoring, they *need* to be pursued. Luckily, there seem to be consistencies in the ways students talk about the concepts of programming and the Hailstone problem (section 3.4). It is actually an advantage that novices (especially those of lower ability) tend to speak in vague terms – I believe this will make concept-only understanding sufficient to carry out many of the observed tutorial dialogue tactics. If it turns out that this is too limiting, I will attempt to move to more advanced sentence-level analysis.

Managing dialogue

The overall structure of the CPP dialogues closely matches the underlying task goal structure, confirming similar conclusions reached by McArthur [30]. This is no surprise – goals are discussed then pursued. Subdialogues generally exist for each part of the four-step pattern (figure 2, p. 13), but vary depending on the dialogue history. If the student has already made it clear they are ready to purse a particular schema (i.e., by suggesting parts of it), then no goal discussion is present. The tutor does, at times, inject a statement indicating the purpose of doing some steps, but this varies as well.

Because I am proposing to build a system that elicits goals and schemas only (no pseudocode), I have focused this discussion on those parts of the CPP dialogues. In general, after a goal is established, the tutor has a list of concepts that need to be mentioned by the student, and the hope is that these answers will emerge from either the top-level questions or by simple tutoring tactics like pumping. As long as concepts remain unmentioned, tutoring for that goal continues. In addition, when students are unable to answer schema-eliciting questions, the use of hand calculations in tutoring works well to elicit desired concepts. Together, these imply different constraints on dialogue management – one that is more open-ended (for top-level goals and schemas) and another that is more specific and predictable (for concrete examples). In section 5 I will describe a *form-filling* approach for top-level dialogue management (to ask the top-level questions) and a *finite-state* approach to the implementation of tutoring tactics.

Regarding turn-taking and initiative, CPP dialogues are generally controlled by the tutor. However, there are a non-trivial number of questions and unsolicited student utterances present in the corpus. During the study, students were aware that a human tutor was on the other end of the interface (it was not a true "Wizard-of-Oz" experiment), and the interface did allow students to type and submit at any time (although the messages appeared only after the submit). The most common form of student initiative was when the student decided s/he had something more to add and wanted to "get it in" before the tutor responded. Recent studies have shown that minimal, if any correlation between mixed-initiative dialogue and learning [9], and thus I will not be proposing any significant steps to address these issues.

4.3 ProPl: An ITS for Novice Program Design

I have chosen to begin by building a prototype system with Atlas [45], a domain-independent toolkit for building natural language tutoring systems. Tutoring is accomplished by leading students through directed lines of reasoning, called *Knowledge Construction Dialogues* (KCDs). A KCD consists of a sequence of tutorial goals, each realized as a question along with an optional explanation. Remediation goals are associated with a set of expected wrong or partially correct answers each of which are associated with directed lines of reasoning as well. Thus, KCDs take on a hierarchical structure and follow a recursive, finite-state based approach to dialogue management. To write a KCD, a author must identify the stages of a directed line of reasoning, edit the textual output the student will see, and predict potential answers to the questions. Atlas also provides an *anything-else* case to handle unrecognized utterances. KCDs seem to be an ideal starting place to implement many of the tutoring tactics found in the corpus, especially those involving concrete examples.

I am calling this system ProPl (from "PROgram PLanner", pronounced "pro-PELL"). It is a partial implementation of CPP but assumes a similar role: ProPl is intended to provide the first exposure of a programming problem to a student. The interface runs through any Java-enabled web browser, and is connected to a back-end (implemented in LISP) that controls the tutoring. To confirm an intuitive understanding, a ProPl tutoring session begins with an interactive hand calculation in the dialogue window. This occurs in dialogue form, just as in CPP. After this, the system engages the student about the problem. In terms of the CPP four-step pattern (figure 2, p. 13), ProPl is able to discuss goals and schemas, but not pseudocode. Therefore, the pseudocode window used in the human-to-human study of CPP has been replaced with a "design notes" window.

A screenshot appears in figure 9. The window in the upper left corner contains a mini-browser for HTML pages that contain the problem statement and any other background information the problem author may choose to provide. The lower left window is for dialogue between the system and the student. Below the dialogue window is a text entry box for student's contributions and three buttons ("I don't know", "Please Explain", and "Please Help"). Currently these buttons are inactive, however their eventual role is to provide the student a way out when s/he is not sure what the system expects. The right half, which held pseudocode in the CPP environment, now holds program design notes. As the dialogue proceeds, goals are established and observations made about how to attain those goals.

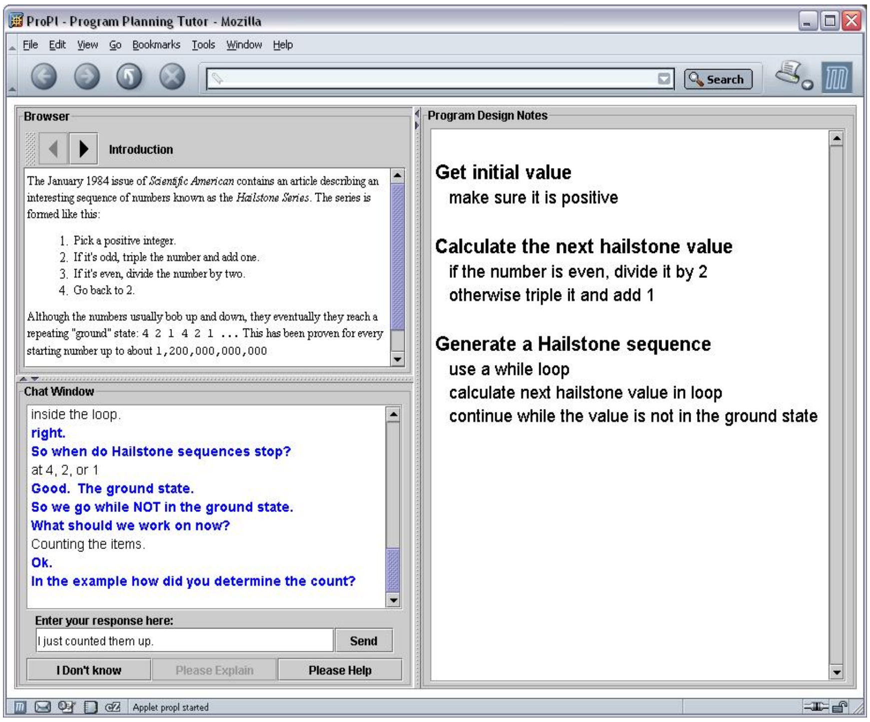

Figure 9: ProPl Environment

When the systems deems it necessary (after a corresponding KCD completes), these notes are recorded in the design notes window. A sample set of design notes for the first three goals of Hailstone appear in the right half of figure 9.

These notes are authored by the domain expert and linked to the dialogue knowledge sources so the system knows which lines to post and when. Thus, for now, the system has complete control over this window. Typically, 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 dialogues). Essentially, this is a record of what the problem author thinks a student should know going into an implementation.

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

Figure 10: Determining the need for a loop

KCDs in ProPl

At this time, Hailstone KCDs have been written to present hand calculations, to elicit programming goals, and to elicit schemas. Some of the example-based KCDs involve only brief references to the examples, while others involve complete runs through entire sequences. All are derived from the corpus as are the predicted answers. To control dialogue, ProPl simply calls KCDs in a fixed order, and thus the resulting dialogue behavior is somewhat rigid. When students are asked about goals, understanding is limited to the list of expected answers in the KCDs. Many of the simple strategies are encoded directly into the KCDs since they typically only use one turn. Atlas includes tools that provide for easy authoring and revising, and thus new data is easily integrated into the knowledge sources.

4.4 Motivation for extending ProPl

In the next section I will be presenting my proposal to extend ProPl to improve its chances of handling the kinds of dialogue behaviors novices display in the CPP corpus. The motivation for doing this stems from the behaviors discussed in section 3.4 that suggested students tend to make good suggestions. The task of the tutor, then, is to recognize and react to these suggestions appropriately.

Figure 10 shows a dialogue from the corpus exhibiting some of the traits that motivate this goal. Here we see a student unsure about how to proceed who looks to the problem statement for help. The suggestion is to count the items (line 1), but comes at a time before the topic of producing a sequence has been discussed. It was likely the student was relying solely on the problem statement by just stating what it said to do next (i.e., count the items). Through a simple line of questioning, the student is led to the observation that a loop is needed.

These exchanges are suggestive of some the reasoning I believe underlies many of the dialogue moves present in my corpus. For example, the student mentions *odd* and *new number* in line 4.[7] The first observation to make is that the student is taking initiative by ignoring the tutor's precise question in line 3 (it is a yes/no question). The second aspect to mention is the inferences that the tutor makes based on the content of the students utterance. These concepts are part of the plan that achieves the *get-next-value* goal, which in turn are part of the *generate-sequence* plan. This reasoning confirms that the student is heading in the right direction. The tutor should respond in an appropriate, and encouraging way, and does in line 5. Similarly, in line 6 the student mentions "the ground state." Even if we ignore the "not" (generally unwise in language understanding), it is still possible to reason that the student is talking about the termination of the loop. Because of this, the tutor senses that the student is "getting close" and that it mention of a generating a whole sequence is drawing near.

Figures 11 shows a student answer that is as close to incorrect as it gets in the CPP corpus. At this point in the dialogue, it is time to start talking about either the counter or

[7] The student accidentally says "positive" instead of "even" in line 4, but this is ignored by the tutor. The number of students who made similar positive/negative for even/odd slips was far greater than I ever would have imagined.

0	T	alright. Let's move on... what should we work on next?
1	S	**What to do when the number does equal 4**

Figure 11: A surprise goal suggestion

0	T	There is only one thing left to do with the counter.
1	T	Do you know what that is?
2	S	**record its value after the loop**

Figure 12: A flawed schema part suggestion

finding the largest value. However, the student seems to believe that values in the ground state are not yet handled properly. The suggestion is to look at when numbers *do* reach the ground state. The student is talking about the termination condition ("equal 4"), and hence still talking about the loop. In the actual dialogue the tutor pursues this misconception in a fairly advanced way (discussing the properties of a Hailstone sequence), but an automated system could reason that the student is still thinking about the loop and can be gently nudged in the direction of thinking about the count, perhaps with a hint to the problem statement, or something similar.

Figure 12 is another example of an unexpected response. In this case, the tutor wants the student to indicate that the final counter value needs to be printed on the screen. The student says to "record" it which is term associated with assignment. While there are many tutorial responses to this, the simplest response would be to suggest the answer is "close" (it does involve the use of a variable) and pump for further description. The hope would be to see a keyword such as "print" or "display" in the student's response.

5 Proposed Research

My research plan consists of extending the prototype version of ProPl and evaluating it in terms of its pedagogical value. At the end of this section, I include a proposed dissertation outline and time table for completion.

5.1 Extensions to ProPl

As discussed in the previous section, ProPl is currently implemented as a set of KCDs called in a fixed order. Help ends with design notes offering nothing specific to address plan merging, something known to be a major source of difficulties for novices [51]. In this section I will first propose a simple extension to better address the issue of plan merging and then suggest an addition to Atlas to improve the robustness of KCD driven dialogues.

Adding pseudocode

Pseudocode plays an important role in the CPP dialogues. It represents the culmination of the goal and schema discussions that occur and gives both the student and tutor something concrete to refer to. Because students are responsible for arranging the steps, they are able to learn about program structure and ordering.

Ultimately I would like to support full pseudocode, however at this time I am proposing to augment the current version of ProPl to simply provide pseudocode once each goal and schema has been identified. Although the student will only feel an indirect responsibility for its creation at best, displaying a fleshed out version of the more abstract goal- and schema- oriented dialogue will act as a sort of cognitive rubber stamp for the prior dialogue. As new goals are introduced and discussed, this pseudocode will grow, showing the new steps in a highlighted form. By doing this, the system is directly addressing the issue of plan merging. For example, a common novice misconception is that multiple loops need to be used when one will do. In Hailstone, this would correspond to implementing one loop for counting and a second (different) loop for finding the largest. In ProPl this sort of misconception is averted by (1) authoring the KCDs to specifically talk about one loop, and (2) by providing pseudocode that grows along with the dialogue. Within the interface, this could be accomplished by introducing a new window that sits below the notes window (as a tab, perhaps). When the tutor is ready to show the student pseudocode, the right half simply changes views. This could be also be accompanied by a short mini-lesson driving home the connection between the pseudocode and previous dialogue.

Extending understanding capability

It is generally believed that failure to understand a student's contribution can have an impact on their affective state. The worst case is a student who becomes so frustrated as to withdraw completely from the system and tutoring session. More commonly, a system's inability to understand simply becomes a distraction. While improving a system's ability to understand seems to be a universal goal for dialogue systems, it seems particularly relevant for novice program design. With the combination of irresolute users and a generally broad range of acceptable answers, the line between right and wrong is sometimes blurred. There are usually many ways to say something productive, or at least not wrong. The novices participating in my research seem particularly adept at providing answers falling into these categories. My proposed extensions to ProPl's dialogue capabilities are motivated by the dialogues in the CPP corpus and are intended to increase its ability to carry out tutorial dialogue more naturally and effectively.

In cases when an answer is not wrong, but not what the tutor was looking for, an automated system must be careful to not classify these as incorrect. Rather, it should attempt to isolate the "grains of truth" and respond appropriately. In some cases, this can be accomplished by inserting an appropriate acknowledgement ("Well, that's true, but...") while in others a short, relevant explanation or subdialogue is necessary. The computational challenge is therefore to:

1. understand a student's utterance, identifying answer aspects
2. situate these concepts within the context of the domain, problem, and solution
3. plan an appropriate response that grounds the students utterance (implicit or otherwise)

As mentioned earlier, it is common for tutorial dialogue systems to require the authoring of expected responses. The problem of how to handle a response outside of the expected answer list is largely un-addressed in these systems, however. In Atlas, the dialogue toolkit being used in ProPl, these are captured by an "anything else" category. KCD authors are encouraged to write "with the intent of sounding natural almost no matter what the student types" [45, p.5]. Thus, there is the possibility that good aspects of unexpected answers will be missed and treated as incorrect. Ideally, a tutor should work with what the student provides as much as possible, especially in more open-ended design dialogue that has a "brainstorming" feel to it.

KCD recovery

I am proposing to extend Atlas to attempt to re-establish a KCD in lieu of following the "anything else" branch. Thus, I seek to go beyond a list of expected answers, thereby broadening ProPl's ability to recognize less obvious aspects of good answers and to respond to them in a way that demonstrates this understanding. The idea to delay following an "anything else" branch until absolutely necessary. In essence, I am proposing a KCD recovery strategy based on the observation that the sources for student utterances are generally predictable (e.g., problem statement, low-level programming primitives, etc.). By having a general representation of these concepts, it may be possible to link them to those that represent expected answers in the KCD. The aim is therefore to *generate dialogue moves based on out-of-scope utterances that return dialogue back to the scope of the KCD.*

I will begin this process by collecting vocabulary novices use to express programming concepts, grouping them by concept. As sources for this I will use the CPP corpus (see section 3.4), data from pilot testing of ProPl, and data from related research.[8] I will then implement some simple dialogue moves such as *acknowledgement* and *pumping* [19], then attempt to identify more elaborate moves If a more elaborate approach to classification seems necessary, such as an ontology or LSA, then I will move in that direction.

Dialogue management and system architecture

Figure 13 shows my proposed architecture for ProPl. I am proposing to build a top-level, *form-based* dialogue manager to handle goals and how to achieve them. A form is needed for each goal and slots represent parts of a schema needed to achieve the goal (along with other concepts deemed necessary by the domain author). The top-level operators are responsible for calling KCDs that achieve the task and communicative goals of the tutor. There is at least one KCD for each goal and schema in the problem, and depending on the problem state (checked by the top-level operator preconditions), an appropriate KCD is selected and fired. During the execution of a KCD, unclassified answers (normally handled an "anything else" dialogue) will be handed to the KCD recovery operators. This will include the context established by the KCD itself (the targeted elicitation(s) and how far along in the KCD the student is) as well as the collection of concepts identified in the KCD.[9] A basic benefit of

[8]A number of researchers have analyzed novice language in an effort to create programming languages that are better cognitive fit for learning [6, 35]. Although the research goals are different from mine, this data does include commonly used vocabulary for concepts such as repetition, selection, mathematical operations, and so on.

[9]It will be important, for example, to know which answers are correct and which are not. It may be unwise to *purposefully* direct the student to an incorrect node, however if the student *happens* to provide

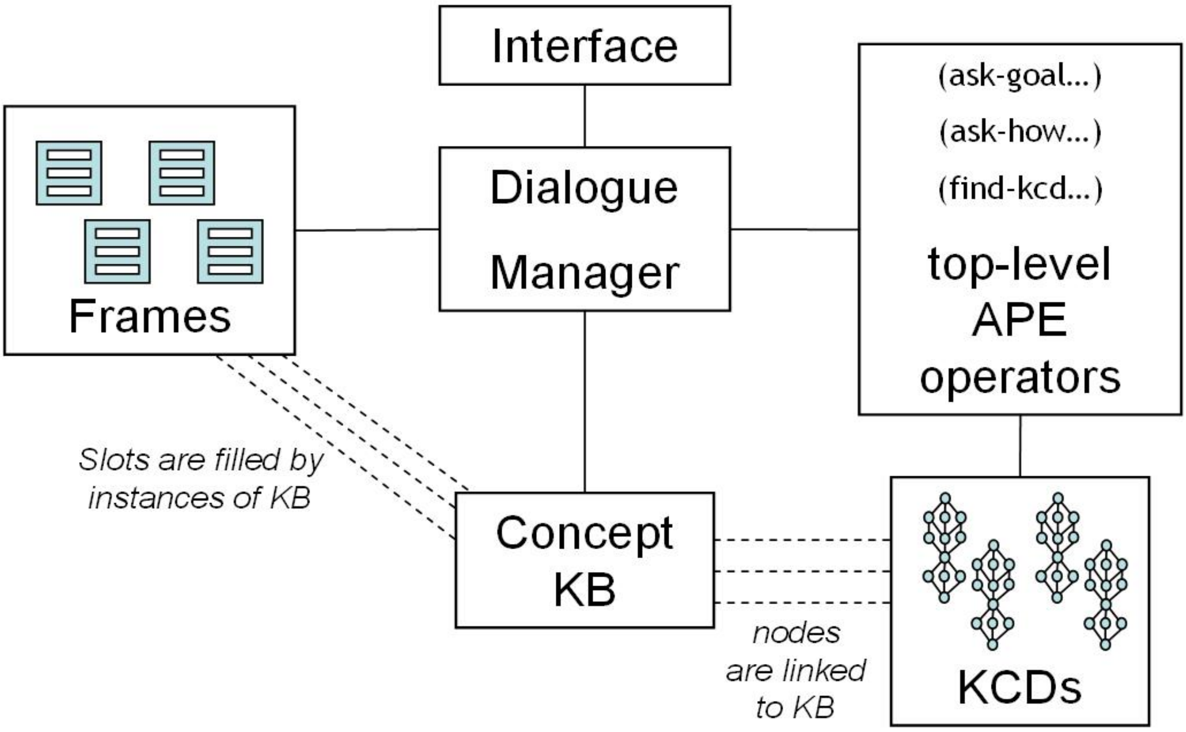

Figure 13: Proposed architecture for ProPl

using a form-based approach to dialogue is that unsolicited information is easier to handle. As such, I also propose to run student utterances through the knowledge base of concepts and any additional (useful) information will be incorporated into the form for that goal.

Remaining questions

There are many issues yet to resolve. My research plan is to build the parts that are fundamental (dialogue manager, top-level operators, keyword KB) and move on as answers emerge for the following questions:

- Is a keyword driven recognizer sufficient?
- What role does the context of the KCD play in making KCD recovery moves?

such an answer, the KCD may be able to pick it up from that point.

- What kinds of reasoning is involved in choosing recovery moves?
- Will a more elaborate form of knowledge representation be necessary, such as an ontology?

It seems likely that many taxonomic relationships could be used to generate some dialogue moves. For example, Milward [33] identifies strategies for asking follow-up questions based on hypernymy, meronymy, and other relations between concepts. I am continuing my study of the CPP corpus to determine if these, and other kinds of relationships, may be useful.

5.2 Evaluation

I aim to elaborate on the general impact of preventive tutoring on novice programming. More specifically, I want determine the effect ProPl has on novices in terms of both their ability to decompose novel problems and on their general programming skills. By using ProPl for multiple assignments, my hope is that novices will (1) learn about how to identify and precisely state programming goals and (2) adopt the general behavior of planning ahead (i.e., thinking about how to achieve the goals they have identified).

The CPP evaluation revealed that students receiving full tutoring with pseudocode significantly outperformed those in a true baseline control group who received no support (other than what is normally available to them (see appendix D). For my proposed evaluation, I intend to narrow this gap by comparing ProPl with a stronger control group that *also* receives pseudocode. By doing this, it becomes more likely that any resulting learning gains are due to tutorial intervention rather than the presence of pseudocode. Because ProPl is an intelligent tutoring system and not a human (as in CPP), this represents a somewhat risky evaluation. To help improve ProPl's chances, I am also proposing to tutor students in this condition on an extra assignment, one that they are not required to implement. This would occur at some point before the post-test and would be done simply to give ProPl students some additional practice identifying goals and schemas.

I plan to recruit subjects from sections of *CS7: Introduction to Programming* at the University of Pittsburgh. Most students who take this course have little or no programming background. Students with more than a semester of programming experience will be turned away. Each subject will be given a questionnaire asking about programming background, their Math SAT score, and attitudes towards programming. In addition, each subject will take a programming pre-test I have devised consisting of a questions testing general programming knowledge and competence. This test will involve both low-level programming specifics, as well as higher-level strategic questions. As in the CPP study, subjects will be paid $7/hour for their participation.

In sum, I am proposing to have two groups: one that uses ProPl and another that does not. Both groups receive pseudocode solutions to their projects before they begin their implementations. The full sequence of events is as follows:

1. both groups take pre-test
2. ProPl subjects are tutored on Hailstone
3. both groups receive pseudocode and proceed to solve Hailstone
4. ProPl subjects are tutored on supplementary problem
5. ProPl subjects are tutored on RPS
6. both groups receive pseudocode and proceed to solve RPS
7. both groups are given post-test programming problem
8. both groups take post-test

The post-test will directly address problem-decomposition skills, the target of the instruction in ProPl. This will help determine the impact ProPl has had on novice strategic knowledge. In addition, I will collect online protocols of compiler and editor usage (as I did in the CPP evaluation) and check traditional measures of success (i.e., compile counts, time spent, and comment quantity). Judging problem decompositions is subjective, therefore I will hire an experienced programming instructor to grade them with me. We will grade each decomposition with a letter grade ('A' through 'F'), and run a kappa to confirm agreement. We will look specifically for the things ProPl was intended to teach, like a clear set of programming goals, statements of how to go about reaching those goals, and any other elements traditionally associated with design.

In addition, I plan to give a small online questionnaire that students will see each time after using ProPl to gauge their immediate feelings on the system (e.g., satisfaction, usefulness, etc.) as well as another with the post-test to gauge general reactions. I will check to see if any of these self-reported measures correlate with any learning gains.

5.3 Dissertation outline

1. Introduction and Contributions
2. Background and Motivation
   (a) Novice programming
   (b) Human tutoring
   (c) Intelligent Tutoring Systems
   (d) Preventive tutoring
3. Tutoring in a Design Domain
   (a) Design-oriented vs. other kinds of tutoring
   (b) Coached Program Planning
   (c) Dialogue and tutoring tactics in CPP
4. Robust Understanding in Dialogue
   (a) Knowledge sources for dialogue systems
   (b) Tutorial dialogue planning
   (c) Knowledge Construction Dialogues
   (d) The use of an ontology in tutorial dialogue
5. ProPl: A System for Novice Program Design
   (a) System design considerations
   (b) Architecture
   (c) Evaluation and Discussion
   (d) Comparison to other programming ITSs
6. Conclusions and Future Work

5.4 Time Line

All points on this time line are targeted *completion* dates, unless otherwise specified.

10.29.2003	Dissertation proposal
11.06.2003	End of fall pilot study of ProPl
11.21.2003	Fall Data analysis & KCD Revisions
11.30.2003	Specifications & requirements for KB
12.05.2003	Finalize literature review, Dissertation chapter 2
12.14.2003	Form-based dialogue manager
12.19.2004	Top-level APE operators
12.31.2003	Testing and debugging of KB, Dissertation chapter 3
01.30.2004	In-house testing and debugging of new ProPl
02.02.2004	Begin spring evaluation, advertise for subjects
02.13.2004	Complete running subjects on Hailstone problem
02.27.2004	Complete running subjects on RPS
03.10.2004	Written post-test
03.26.2004	Programming post-test (on-line protocols)
03.31.2004	Dissertation chapter 4
04.23.2004	Analysis of Spring data
05.17.2004	Dissertation chapter 5
05.24.2004	Dissertation chapters 1 and 6, Draft complete
06.16.2004	Individual meetings with committee members
06.25.2004	Finalize dissertation, distribute to committee
07.16.2004	Dissertation defense

6 Contributions

I have undertaken the challenges of studying the novice programmer and of building a system that seeks to alleviate some of their difficulties. As such, I feel my research makes contributions along both of these dimensions. My research consists of both novel applications of existing methods and of unique contributions.

Improved understanding of novice programmers. Much work has been done studying the programs produced by novices [25, 28, 48, 52]. In these studies, it is generally necessary to infer the original intent of the student based on the program code. Researchers have also studied the cognitive happenings of novices as they write code [40], sometimes with the help of verbal protocols [38]. My work contributes to this stream by looking at novices in a *design context*. In both CPP and ProPl, students are discouraged from talking about programming language specific details. What impact does this cognitive "relocation" have on a novice? What is the relationship between misconceptions that arise during the design phases of programming with those that arise during a typical, unplanned implementation? Another interesting aspect involves cultural and behavioral change. Is it possible to affect behaviors and views of programming through the use of preventive tutoring? By building this system for novices, and basing it on human-to-human tutoring, some answers to these questions should naturally arise (I believe some already have).

A more robust approach for tutorial dialogue systems. Driven by real human data, I have proposed to extend an already elaborate framework for natural language tutoring (i.e, Atlas) so that it can be more responsive in the face of unexpected student input. The programming domain provides fertile ground for such an approach due to the seemingly open-ended nature of design. If successful, an ability to re-establish the line of reasoning in a KCD by using generalized planning operators will represent a significant step in the goal of broadening the applicability of the KCD approach. Although I have chosen to focus specifically on programming, my hope is that the general strategy will be applicable beyond this domain (although I am not proposing to test this hypothesis in my dissertation).

Cognitive "graduation" and the use of examples in tutoring. It is generally acknowledged that examples are appealing to students. GIL [43] works from this assumption, but using them at an extremely fine-grained level. My approach is to use examples to first connect with students intuitive notions by doing hand calculations and following this with targeted, simple lines of questioning. This questioning is used for two main reasons: (1) to help the student see hidden details that might otherwise be glossed over, and (2) to help the student begin to see when intuitive strategies are and are not useful. It is serendipitous that KCDs are perfectly suited for such dialogues, which is why I chose to build ProPl with them examples as the tutorial centerpiece.

Practical benefits for teachers of programming. Traditional methods of distributing an assignment (e.g., handout or web page) are passive and engaging only if the student who choose to embrace the importance of understanding and planning (*very* rare). My system introduces problems interactively and promotes the early confrontation of difficult programming choices. It attempts to reveal hidden intricacies of tasks that otherwise may seem easy to novices. If successful, and a large enough library of problems can be authored, I believe it could be of interest to many introductory programming teachers. Indeed, I have received numerous emails inquiring about the status of the project and spoke with many more at conferences.

Reification of schema/plan knowledge. The issue of whether Soloway-style plan knowledge should be reified has not been resolved. My work challenges the assumption that it *should* be. My approach is to base as much as possible on the existing knowledge of the student (including the use of natural language) using a cognitive load argument as support. When a students incoming knowledge or intuitive understanding of a task might present an impasse, I have chosen to use tutoring as a means for pointing out the futility of following such instincts. The knowledge underlying problem solutions in my proposed system is represented in terms of programming goals, schemas, and plans. The stance I take is that it is not necessary to reify this knowledge, but rather help the student self-construct through the use of accessible and intuitive notions.

Intelligent tutoring in a design domain. KCDs have not yet been applied to a design-oriented task domain, and thus the issues that arise and changes they suggest to Atlas will be of interest to those working in the area of tutorial dialogue systems. The general goal structure of the dialogues, which for CPP is driven by the four-step pattern, may be able to be generalized for any design task. When designing from scratch, discussion of goals and how to achieve them abstractly seem unavoidable.

Easier authoring of tutorial dialogue knowledge sources. Writing KCDs is often tedious, requiring cycles of revision. If my work is successful, some of this burden will be shifted to the creation of an ontology. The benefit is that once an ontology for a domain is complete, domain authors can worry less about the completeness of their expected answer lists. The introduction of problem specific knowledge and solution knowledge is also required, but conceivably, much of this could be automated (although I do not intend to pursue this in my dissertation).

References

[1] Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences 4*, 2 (1995), 167–207. [www].

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

[3] Bonar, J. G., and Cunningham, R. Bridge: Tutoring the programming process. In *Intelligent Tutoring Systems: Lessons Learned*, J. Psotka, L. D. Massey, and S. A. Mutter, Eds. 1988, 1988, pp. 409–434.

[4] Bonar, J. G., and Soloway, E. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interactions 1*, 2 (1985), 133–161.

[5] Brown, D. E. Using examples and analogies to remediate misconceptions in physics: Factors influencing conceptual change. *Journal of Research in Science Teaching 29*, 1 (1992), 17–34.

[6] Bruckman, A., and Edwards, E. Should we leverage natural-language knowledge? an analysis of user errors in a natural-language-style programming language. In *Proceedings of the Conference on Human Factors in Computing Systems* (Pittsburgh, PA, 1999), pp. 207–214.

[7] Chi, M., Feltovich, P. J., and Glaser, R. Categorization and representation of physics problems by experts and novices. *Cognitive Science 5* (1981), 121–152. [www].

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

[9] Core, M. G., Moore, J. D., and Zinn, C. The role of initiative in tutorial discourse. In *10th Conference of the European Chapter of the Association for Computational Linguistics (to appear)* (Budapest, Hungary, 2003). [www].

[10] de Lemos, M. A., and de Barros, L. N. A didactic interface in a programming tutor. In *Proceedings of 11th International Conference on Artificial Intelligence in Education (AIED2003)* (2003). [www]. Innovations in Teaching Programming Workshop.

[11] Deek, F. P. A framework for an automated problem solving and program development environment. *Journal of Integrated Design and Process Science 3*, 3 (1999), 1–13. [www].

[12] DuBoulay, B. Some difficulties of learning to program. *Journal of Educational Computing Research 2*, 1 (1986), 57–73.

[13] Fay, A. L., and Mayer, R. E. Benefits of teaching design skills before teaching logo computer programming: Evidence for syntax independent learning. *Journal of Educational Computing Research 11*, 3 (1994), 187–210.

[14] Feddon, J. S., and Charness, N. Component relationships depend on skill in programming? In *11th Annual PPIG Workshop* (University of Leeds, UK, 1999). [www].

[15] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object- Oriented Software*. Addison Wesley, 1995. [www].

[16] Gegg-Harrison, T. S. Exploiting program schemata in an automated program debugger. *Journal of Artificial Intelligence in Education 5*, 2 (1994), 255–278.

[17] Glass, M. Processing language input in the circsim-tutor intelligent tutoring system. In *Artificial Intelligence in Education* (2001), J. Moore, Ed., IOS Press, pp. 210–221. [www].

[18] Glass, M., Kim, J. H., Evens, M. W., Michael, J. A., and Rovick, A. A. Novice vs. expert tutors: A comparison of style. In *Tenth Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 99)* (Bloomington, IN, 1999), pp. 43–49. [www].

[19] Graesser, A. C., Person, N. K., and Harter, D. Teaching tactics and dialog in autotutor. *International Journal of Artificial Intelligence in Education 12* (2001), 257–279. [www].

[20] Graesser, A. C., Person, N. K., and Magliano, J. P. Collaborative dialogue patterns in naturalistic one-to-one tutoring. *Applied Cognitive Psychology 9* (1995), 495–522.

[21] Graesser, A. C., Wiemer-Hastings, P., Person, N. K., and Harter, D. Using latent semantic analysis to evaluate the contributions of students in autotutor. *Interactive Learning Environments 8* (2000), 129–148. [www].

[22] Guzdial, M. Programming environments for novices. In *Handbook of Computer Science Education Research*, S. Fincher and M. Petre, Eds. Springer–Verlag, 2003 (to appear). [www].

[23] Guzdial, M., Hohmann, L., Konneman, M., Walton, C., and Soloway, E. Supporting programming and learning-to-program with an integrated cad and scaffolding workbench. *Interactive Learning Environments 6*, 1&2 (1998), 143–179.

[24] Heffernan, N. T. *Intelligent Tutoring Systems have Forgotten the Tutor: Adding a Cognitive Model of Human Tutors*. PhD thesis, Carnegie-Mellon University, March 2001. [www]. Tech Report CMU-CS-01-127.

[25] Johnson, W. L. Understanding and debugging novice programs. *Artificial Intelligence 42* (1990), 51–97.

[26] Joni, S.-N., and Soloway, E. But my program runs! discourse rules for novice programmers. *Journal of Educational Computing Research 2*, 1 (1986), 95–125.

[27] Lane, H. C., and VanLehn, K. Coached program planning: Dialogue-based support for novice program design. In *Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE)* (Reno, NV, 2003), pp. 148–152. [www].

[28] Littman, D., Soloway, E., and Pinto, J. The knowledge required for tutorial planning: An empirical study. *Interactive Learning Environments 1*, 2 (1990).

[29] Mayer, R. E., Ed. *Teaching and Learning Computer Programming.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[30] McArthur, D., Stasz, C., and Zmuidzinas, M. Tutoring techniques in algebra. *Cognition and Instruction 7*, 3 (1990), 197–244.

[31] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulliten 33*, 4 (2001), 125–140. [www]. Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-year CS.

[32] McKendree, J., Radlinski, B., and Atwood, M. E. The grace tutor: A qualified success. In *Intelligent Tutoring Systems: Second International Conference, ITS '92 Proceedings* (Berlin, 1992), C. Frasson, G. Gauthier, and G. I. McCalla, Eds., Springer-Verlag, pp. 677–684.

[33] Milward, D., and Beveridge, M. Ontology-based dialogue systems. In *Third Workshop on knowledge and reasoning in practical in practical dialogue systems (IJCAI '03)* (Acapulco, Mexico, 2003). [www].

[34] Newell, A., and Simon, H. *Human Problem Solving.* Prentice Hall, 1972.

[35] Pane, J. F. *A Programming System for Children that is designed for Usability.* PhD thesis, Carnegie Mellong University, Pittsburgh, PA, 2002. CMU-CS-02-127.

[36] Pea, R. D., and Kurland, M. D. On the cognitive prequisites of learning computer programming. Tech. Rep. 18, Bank Street College of Education, New York, NY, 1983.

[37] Pennington, N. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Ablex Corp., Norwood, New Jersey, 1987, pp. 100–113.

[38] Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. Conditions of learning in novice programmers. In Soloway and Spohrer [49], pp. 261–279.

[39] Perkins, D. N., Schwartz, S., and Simmons, R. Instructional strategies for the problems of novice programmers. In Mayer [29], pp. 153–178.

[40] Pintrich, P. R., Berger, C. F., and Stemmer, P. M. Students' programming behavior in a pascal course. *Journal of Research in Science Teaching 24*, 5 (1987), 451–466.

[41] Polya, G. *How to Solve It*. Doubleday–Anchor, New York, 1957.

[42] Ramadhan, H. A., Deek, F., and Shihab, K. Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Artificial Intelligence Review 16* (2001), 61–84.

[43] Reiser, B. J., Kimberg, D. Y., Lovett, M. C., and Ranney, M. Knowledge representation and explanation in gil, an intelligent tutor for programming. In *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*. J. H. Larkin and R. W. Chabay, 1992, pp. 111–150.

[44] Rist, R. S. Schema Creation in Programming. *Cognitive Science 13* (1989), 389–414.

[45] Rose, C. P., and Jordan, P. Interactive conceptual tutoring in atlas-andes. In *Proceedings of AI in Education 2000 Conference* (2000).

[46] Soloway, E. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM 29*, 9 (September 1989), 850–858. [www].

[47] Soloway, E., Bonar, J., and Ehrlich, K. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM 26*, 11 (1983), 853–860. [www].

[48] Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. What do novices know about programming? In *Directions in Human-Computer Interaction*, A. Badre and B. Schneiderman, Eds. Ablex Corp., Norwood, New Jersey, 1982, pp. 87–122.

[49] Soloway, E., and Spohrer, J. C., Eds. *Studying the Novice Programmer*. Ablex Corp., Norwood, New Jersey, 1989.

[50] Soloway, E., Spohrer, J. C., and Littman, D. E unum pluribus: Generating alternative designs. In *Teaching and Learning Computer Programming*, R. E. Mayer, Ed. 137–152, 1988.

[51] Spohrer, J. C., and Soloway, E. Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics* (Tucson, Arizona, November 12-15 1985).

[52] Spohrer, J. C., and Soloway, E. Novice mistakes: Are the folk wisdoms correct? In Soloway and Spohrer [49], pp. 401–416. [www].

[53] Sweller, J. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction 4* (1994), 295–312.

[54] Weber, G., and Brusilovsky, P. Elm-art: An adaptive versatile system for web-based instruction. *International Journal of Artificial Intelligence in Education 12* (2001), 351–384. [www].

[55] Wender, K. F., Weber, G., and Waloszek, G. Psychological considerations for the design of tutorial systems. In *Proceedings of the Third International Conference on Artificial Intelligence and Education* (Pittsburgh, PA, 1987).

A The Hailstone Problem

The *Hailstone* is an example of a knowledge-lean programming assignment typically given four to five weeks into an introductory programming course. 14.[10]

> 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 14: The Hailstone Problem

For most novices, producing a sequence according to the rules is trivial as is counting them and finding the largest. In the corpus, every student was able to do a hand calculation with ease. However, writing a program for Hailstone is very challenging. The solution 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. Nonetheless, Hailstone has a fairly straightforward goal decomposition:

1. Get initial value
2. Calculate next value
3. Generate a sequence
4. Count the items

[10] I first encountered this problem in Doug Cooper's popular textbook *Oh Pascal!*.

```
get a positive number from the user
initialize count to one
initialize largest to the input
while number generated is not 4 do
   if the number is odd then
         multiply it by 3 and add one
   else
         divide the num by 2
   increase count by 1
   if current value is larger than largest then
         store the new largest value
end while
print count
print largest
```

Figure 15: pseudocode solving the Hailstone problem

5. Find the largest

Two of these goals are given directly in the problem statement (counting and finding the largest), while the others can be inferred from the steps. While other decompositions are certainly imaginable, this is the one most-often promoted by the tutor in our study. A pseudocode solution resulting from a CPP tutoring session appears in figure 15.

A student employing a staged approach to Hailstone might write a program that obtains user input and produces the next value in the sequence. Once this program works, the next step would be to add a loop to generate a sequence. Similarly, that result would be augmented to count, and finally to track the largest.

B Problems used in the CPP study

The corpus consists of 48 dialogues covering 5 different problems. The problems are as follows:

1. **Hailstone problem**: see appendix A
2. **Rock-Paper-Scissors (RPS)**: play a multiple game match with the user; program chooses randomly, keeps track of win/loss record, and ends when a majority of games have been won (i.e., "best of" or over half); involves functions and enumerated types
3. **Number to words (NTW)**: program should input a number between 1 and 999 then display the number in English words (e.g., "Four hundred eighty three"); involves procedures;
4. **Weather calculations:** requires a menu-driven program allowing the program user to choose a weather calculation (e.g., wind-chill factor), provide necessary input, and display the result; involves functions
5. **Baseball statistics**: input is raw baseball data; task is to output common statistics, like batting average, slugging percentage, etc.; involves file i/o

The first three are knowledge-lean while the final two are not. Both the weather and baseball problems were given during pilot phases of CPP and were awkward to tutor. The basic problem was that the hand calculations were essentially students' best estimates of desired program behavior. Also, the solutions required more busywork than anything else, involving very little insight or creativity to solve. In sum, they were boring assignments and ill-suited for CPP.

C Example tutoring session

#	Sp	time	utterance
1	T	13:03:36	Just click the done button when you are ready to start.
2	T	13:03:59	let's do a quick example.
3	T	13:04:08	If we start at 12 what comes next in the hailstone sequence?
4	S	13:04:23	**6**
5	T	13:04:27	good. after that?
6	S	13:04:29	**3**
7	T	13:04:43	yep! Next?
8	S	13:04:47	**10**
9	T	13:04:52	ok...
10	S	13:04:53	**5**
11	T	13:05:00	keep it rolling...
12	S	13:05:05	**16**
13	S	13:05:08	**8**
14	S	13:05:10	**4**
15	S	13:05:13	**2**
16	S	13:05:14	**1**
17	T	13:05:20	good.
18	T	13:05:23	what's after 1?
19	S	13:05:28	**4**
20	T	13:05:38	right so this is the ground state. No need to go forward.
21	T	13:05:47	ok so how many elements were in the sequence?
22	S	13:06:18	**11**
23	T	13:06:37	uh oh... better try again. Remember count the initial value and the first value you see in the ground state only.
24	S	13:07:22	**8**
25	T	13:07:36	right! The sequence you counted was 12 6 3 10 5 16 8 4
26	T	13:07:40	8 elements.
27	T	13:07:45	ok what was the largest?
28	S	13:07:53	**16**
29	T	13:07:57	good.
30	T	13:08:14	Ok I think we're ready to write some pseudocode. Is that ok? Or do you want another example?
31	S	13:08:22	**we can go on**
32	T	13:08:26	cool.
33	T	13:08:43	alright what is the first thing our program should do?
34	S	13:08:59	**read an integer from a user**
35	T	13:09:14	good. What variable should we store it in?
36	S	13:09:23	**num**
37	T	13:09:26	sounds great.
38	T	13:09:39	*(tile created: read an integer from a user into num)*
39	S	13:09:47	*(tile placed)*

#	Sp	time	utterance
40	T	13:10:04	you might want a little space there. could you bring it down a line or two and to the right a bit?
41	S	13:10:13	*(tile placed)*
42	T	13:10:16	that's great.
43	T	13:10:22	ok what next?
44	S	13:10:37	**we need to tell if the integer is positive**
45	T	13:11:04	that's right but I think for our pseudocode we'll just leave that out. Let's assume that.
46	S	13:11:09	**ok**
47	S	13:11:23	**then we need to tell if the integer is even or odd**
48	T	13:11:31	it will be important in a real program though. So good thinking.
49	T	13:11:42	ok even or odd. Right. Which one should we handle first?
50	S	13:11:54	**odd**
51	T	13:12:03	ok!
52	T	13:12:09	*(tile created: if num is odd then)*
53	S	13:12:18	*(tile placed)*
54	T	13:12:29	and what do we do to num if that is true?
55	S	13:12:40	**mulitply by 3**
56	S	13:12:42	**then add 1**
57	T	13:13:07	*(tile created: multiply num by 3 and add 1)*
58	S	13:13:15	*(tile placed)*
59	T	13:13:32	if the number is not odd then it must be even.
60	T	13:13:35	*(tile created: else)*
61	S	13:13:37	**right**
62	S	13:13:43	*(tile placed)*
63	T	13:13:49	and in that case?
64	S	13:13:56	**divide by 2**
65	T	13:14:06	*(tile created: divide num by 2)*
66	S	13:14:12	*(tile placed)*
67	T	13:14:34	ok. I think we are getting somewhere. What should we do next?
68	S	13:15:01	**then we need to know if the answer is positive or negetive and start the loop again**
69	T	13:15:21	well... do you think either of those operations can give you a negative result?
70	S	13:15:45	**not negetive i meant even or odd sorry**
71	T	13:15:59	that's ok. we'll get it figured out...
72	T	13:16:18	actually we've already done the even/odd distinction.
73	T	13:16:24	But I like the loop idea a lot!
74	S	13:16:45	**as long as the final number isn't four we can go back up to see if it's odd the beginning of the loop right?**
75	T	13:16:57	very good!
76	T	13:17:09	Ok so we need to incorporate a loop and we will keep going as long as num is not 4.
77	T	13:17:18	what kind of loop do you want to use?
78	S	13:17:51	**the one where you make the statement first like as long a num is not equal to 4 do**
79	S	13:17:57	**a while loop!**

#	Sp	time	utterance
80	T	13:18:05	perfect.
81	T	13:18:22	ok so here's the while tile with your condition inside. You may need to move some tiles around.
82	S	13:18:28	**ok**
83	T	13:18:35	*(tile created: while num is not equal to 4 do)*
84	S	13:18:47	*(tile placed)*
85	T	13:20:00	good. could you move the multiply tile back in a line?
86	S	13:20:08	*(tile placed)*
87	T	13:20:23	and lastly in pseudocode it is a good idea to be clear on the end of loops. So here's that tile.
88	T	13:20:27	*(tile created: end while)*
89	S	13:20:34	*(tile placed)*
90	T	13:20:45	looking good. At this point let's take a step back and think about what we've got.
91	T	13:21:02	Can you tell me what this code accomplishes? in three or four words maybe?
92	S	13:21:46	**breaks down a number using Hailstone Series**
93	T	13:22:02	pretty good! In other words it generates a hailstone series!
94	T	13:22:17	Ok now go ahead and type that into the "description" box and click the snapshot button.
95	T	13:22:26	we are taking a picture of the code as it stands now.
96	T	13:22:59	awesome. Go ahead and click on stage 2 and we'll keep going.
97	T	13:23:16	FYI- in a real program you'd need to put a print statement in the loop to actually see it on the screen.
98	T	13:23:21	does that make sense?
99	S	13:23:35	**yes**
100	T	13:23:43	great.
101	T	13:23:57	what should we work on now?
102	T	13:30:07	(*network lag*) now we're ready to go on. What should we work on next?
103	S	13:30:20	**we need to count how many numbers were in the sequence**
104	T	13:30:28	right. Any ideas on how to do that?
105	T	13:30:34	what step we should add?
106	S	13:30:44	**not sure...**
107	T	13:31:05	that's ok... in our previous example that started with 12 how did you count that one?
108	S	13:31:40	**i just went up and started with 12 and counted how many numbers were in the series until 4 was reached**
109	T	13:31:52	good. so when you started with 12 what was the count at?
110	S	13:31:58	**0**
111	T	13:32:05	well... we do count it as part of the series.
112	T	13:32:06	so 1.
113	S	13:32:14	**oh yeah**
114	S	13:32:14	**right**
115	T	13:32:21	and after that was 6. What's the count now?
116	S	13:32:26	**2**
117	T	13:32:32	good. next is 3. count?
118	S	13:32:36	**3**

#	Sp	time	utterance
119	T	13:32:45	good. Do you see the algorithm yet?
120	T	13:32:54	what are we doing for each hailstone value?
121	S	13:33:04	**we are adding 1 to the count**
122	T	13:33:11	good. And how often do we do that?
123	S	13:33:38	**everytime we go through the loop**
124	T	13:33:58	right! One time through the loop corresponds to one new hailstone value.
125	T	13:34:03	That is a crucial observation.
126	T	13:34:18	ok so we need a variable for this how about count?
127	S	13:34:23	**sounds good**
128	T	13:34:30	what step should we add to the program then?
129	S	13:35:09	**count + 1 everytime we go through the loop**
130	T	13:35:29	right. We need to update the count by adding one to it.
131	T	13:35:35	*(tile created: add 1 to count)*
132	S	13:35:50	*(tile placed)*
133	S	13:36:12	**should it go in the loop?**
134	T	13:36:21	what do your instincts tell you?
135	S	13:36:29	**yes but i'm not sure where**
136	T	13:36:47	good. It definitely goes in the loop. we want to count each time through.
137	S	13:36:48	*(tile placed)*
138	T	13:36:55	now does it have anything to do with the if/else?
139	S	13:37:06	*(tile placed)*
140	S	13:37:09	**no**
141	S	13:37:17	*(tile placed)*
142	T	13:37:23	good. It really doesn't no matter what you see (even or odd) you always add 1 to the count.
143	S	13:37:24	*(tile placed)*
144	T	13:37:27	that looks great.
145	S	13:37:29	*(tile placed)*
146	T	13:37:50	alright we have a little more to do with the count.
147	T	13:37:56	any ideas?
148	S	13:38:06	**well we have to print it at the end**
149	S	13:38:09	**right?**
150	T	13:38:09	yep!
151	T	13:38:14	*(tile created: print count)*
152	S	13:38:20	*(tile placed)*
153	T	13:38:32	good. So that takes care of after the loop. What about before?
154	S	13:39:01	**do we need to print it inside the loop?**
155	T	13:39:08	not really necessary.
156	S	13:39:17	**i didn't think so**
157	T	13:39:20	the problem statement says only to print the final count.
158	T	13:39:30	we've worked inside and after the loop what's left?
159	S	13:39:39	**..**
160	T	13:39:44	how about before the loop?
161	S	13:39:54	**oh we need to establish the variable**
162	T	13:40:06	what do you mean by "establish"?
163	S	13:40:12	**assign**

#	Sp	time	utterance
164	T	13:40:26	when this happens before a loop you call it an initialization.
165	S	13:40:32	**ok**
166	T	13:40:37	what value should we initialize it to?
167	S	13:41:04	**but we need to inclue the intial value in it the value entered by the user so should we start at 1?**
168	T	13:41:16	good. Just like you did in your head during the example.
169	T	13:41:35	Much of programming is taking what you do automatically and breaking it down into component pieces just like this.
170	T	13:41:55	*(tile created: assign 1 to count)*
171	T	13:42:06	you'll need to move the entire loop down some to fit it in.
172	S	13:42:19	*(tile placed)*
173	S	13:42:40	**should i move that up one?**
174	T	13:42:55	move what up one?
175	S	13:43:13	**should i put the assign 1 to count up at the top**
176	T	13:43:31	well it's your pseudocode but I can tell you that it doesn't matter.
177	S	13:43:36	**ok**
178	T	13:43:50	since the two statements are independent they can be done in either order.
179	T	13:43:58	I like what you have if you do?
180	S	13:44:07	**yeah it looks good**
181	T	13:44:25	alright it's time for another snapshot. Considering the code we added in this stage what does it accomplish?
182	S	13:44:48	**counts number in sequence**
183	T	13:45:00	good. go ahead and enter that as the description and do the snapshot.
184	T	13:45:30	alright the last stage awaits. What is left to do?
185	S	13:45:44	**we need to print the largest number**
186	T	13:45:56	good. any ideas on how to find the largest?
187	S	13:46:11	**no none that come to mind right away**
188	T	13:46:25	how did you do it in the example we did earlier?
189	S	13:47:06	**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**
190	T	13:47:20	That's a good idea.
191	T	13:47:37	so we need a new variable.
192	T	13:47:41	what should we call it?
193	S	13:47:48	**large**
194	T	13:47:52	how about largest?
195	S	13:47:54	**ok**
196	T	13:48:04	and what should we initialize largest to?
197	S	13:48:22	**0**
198	T	13:48:34	that could work... but there's something better.
199	S	13:48:34	**or the number orginally entered by the user**
200	T	13:48:37	yes!!
201	T	13:48:45	you thought of that before I said anything. good job.
202	T	13:48:53	ok so in the program what step should we add?
203	S	13:49:09	**assign num to largest**

#	Sp	time	utterance
204	S	13:49:11	**first**
205	T	13:49:26	good. you'll need to make some space again.
206	S	13:49:43	*(tile placed)*
207	T	13:49:47	*(tile created: assign num to largest)*
208	T	13:49:50	*(tile placed)*
209	T	13:50:33	alright and what should we do inside the loop?
210	S	13:51:26	**we should say if num is ¿ largest then we assign that in else we keep the orginal number in there**
211	T	13:51:45	pretty close. Here's the if part.
212	T	13:51:57	*(tile created: if num is ¿ largest then)*
213	S	13:52:12	*(tile placed)*
214	T	13:52:29	and what do you think we should do if that is true?
215	S	13:52:44	**assign num to largest**
216	T	13:52:51	good.
217	T	13:52:52	*(tile created: assign num to largest)*
218	S	13:53:00	*(tile placed)*
219	T	13:53:06	in other words we've found a new largest so update largest to be the current hailstone number.
220	T	13:53:20	and the last thing to add?
221	T	13:53:34	after the loop?
222	S	13:53:44	**print largest**
223	T	13:53:47	*(tile created: print largest)*
224	S	13:53:53	*(tile placed)*
225	T	13:54:16	one little cosmetic adjustment here... the end while (and the two prints) should line up with the while way above it.
226	T	13:54:30	or else they are lined up with the BODY of the loop.
227	S	13:54:33	*(tile placed)*
228	T	13:54:48	that's nice. Ok so go ahead and give a description for the code we added and take the snapshot.
229	S	13:55:11	**determines largest in sequence?**
230	T	13:55:17	sure!
231	T	13:55:33	that's it! Good job.
232	S	13:55:37	**thanks**

D An Evaluation of CPP

I have evaluated CPP in terms of its immediate and longer term impact on student programming behaviors. Subjects (N = 16) in the study were split into a baseline control group receiving no experimental interventions and another that received human-to-human, computer-mediated CPP. An earlier non-trivial programming assignment was used as a pre-test to confirm equivalent competencies of conditions ($t(12) = -0.269$, $p = 0.79$). All subjects did the same assignments. Students in the CPP condition were tutored in two consecutive assignments (Hailstone and RPS), and then went untutored on a third (an assignment requiring the searching and processing of arrays, this acted as the post test). We collected online protocols consisting of editor and compiler logs, including copies of all programs submitted to the compiler.[11]

Several different measures were used to gauge the immediate impact of CPP on students. We looked at the number of indentation errors, the number of compiles, the number of those that were done to fix logical errors (i.e., relating to the underlying algorithm), and lastly, based on editor and compiler timestamps, the amount of time in minutes spent fixing logical errors. The results for the tutored Hailstone project appear in table 3. All measures of success reached significance. Although CPP students outperformed untutored students, it remains unclear which aspects of CPP deserve credit. It could be the interactions in the dialogue, the pseudocode, or perhaps a combination of both.

measure	*CPP Mean (std dev)*	*Ctrl Mean (std dev)*	*Effect Size (sig)*
Indentation mistakes	2.43 (2.88)	12.7 (10.8)	0.95 ($p < 0.04$)
Total number of compiles	14.9 (9.5)	41.4 (38.4)	0.69 ($p < 0.04$)
Compiles to fix logical errors	5.4 (5.0)	25.9 (28.0)	0.73 ($p < 0.01$)
Minutes spent fixing logical errors	15.0 (10.4)	125 (130)	0.85 ($p < 0.03$)

Table 3: CPP results on tutored project (Hailstone)

To see if there were any extended benefits of going through CPP, we investigated an untutored post-test project. The hope was that if we saw differences here, then it was more likely that CPP tutoring was having a more lasting effect on students' behaviors and tendencies.

To do this, we came up more revealing measures of success. The first was to look at the quantity of comments. The idea is that if students learn to think about programs in their

[11] This follows the methodology employed by Spohrer and Soloway [51]. IRB approval was obtained and students signed consent forms before the study.

measure	*CPP Mean (std dev)*	*Ctrl Mean (std dev)*	*Effect Size (sig)*
Number of comment lines	35.0 (23.1)	12.3 (9.76)	2.33 ($p < 0.04$)
"Successful" compiler attempts	25.0 (32.9)	49.4 (41.4)	not sig.
Floundering episodes	1.00 (2.24)	2.71 (2.63)	0.65 ($p < 0.07$)

Table 4: CPP results on untutored (post test) project

own language via CPP, then this mindset may carry through to later assignments, surfacing as a behavior to be more prone to add comments. The second was to look at compiles that completed without syntax errors – a poor man's way to gauge algorithmic difficulties. Lastly, we undertook the arduous task of analyzing all syntactically correct programs submitted to the compiler (side-by-side) and coded the activities happening between each pair. Of most interest were those pairs indicative of *floundering*, defined in this context as an attempted repair of a bug that leaves the program no closer to correct. A floundering *episode* is then defined as one or more of these compiles in succession to fix the same bug. Because the floundering is a subjective measure, the data was coded by two experienced programming instructors. Each compile attempt was classified and intercoder reliability was computed with a +/- 1 cushion on the boundaries of floundering episodes. The result was a kappa value of 0.872.

On the untutored post test project, we found that students who underwent CPP, compared to those who did not, were more prolific with comments in their programs and exhibited less erratic programming behavior during their implementation in terms of floundering episodes. Although the difference for number of error-free compiles was not significant, the floundering result was marginally significant. Considering that the CPP "training" amounted to only two hours on average for each student (an hour each for Hailstone and RPS), an entire semester receiving CPP would (hopefully) reveal even stronger influence. The full details of this analysis can be found in [27].