JOURNAL OF MATHEMATICAL BEHAVIOR **10**, 17–53 (1991)

Learning the Structure of Recursive Programs in Boxer

MICHAEL LEONARD
University of California, Berkeley

Recursive programming is notoriously difficult to learn for adults as well as children. Boxer is a new programming language and environment whose structure, including its structure of recursive programs, has been designed to be concrete and accessible to students. This study explores how one eighth-grade girl learned about the structure of recursive programs in Boxer. A curriculum was developed in Boxer to teach the structure of recursive programs based on the "copy and execute" model of procedure execution. Working from an initial model of recursive programs that had serious flaws, the student made considerable progress towards the intended model. In the process, she successfully analyzed and synthesized many recursive programs. I used impasse resolution both as an analytical frame for understanding her learning and as the basis for an empirical method of analyzing learning. I charted the evolution of her knowledge of recursive program structure in fine detail by analyzing impasses that arose in her problem solving. She was observed to construct new knowledge for herself by overcoming impasses. In situations where she might have learned something, but where an impasse did not arise, the observer did not see her learn. The method of impasse analysis is presented in detail, and its implications for modelling students' programming knowledge, for evaluating Boxer, and for teaching programming are discussed.

1. INTRODUCTION

In March of 1989, when I had been a member of the Boxer research group at U.C. Berkeley for 3 months, I had spent about a day programming in Boxer. I had created a tool for graphing points, tables, and functions, and I had included a little game where students could enter equations and have them graphed in order

I want to thank Steve Adams, Andy diSessa, Masako Itoh, Yuji Itoh, Marcia Linn, Don Ploger, and Alan Schoenfeld for many helpful comments on drafts of this article, the Functions research group, the Boxer research group, and the EMST–SESAME community for providing a helpful audience, Deborah Sanders for editing assistance, Tina Kolpakowski and her students for the opportunity to work with them, and Jane and Virginia for constant support. Opinions and errors in this article are my sole responsibility. An earlier version of this article was presented at the American Educational Research Association Annual Meeting, 1990.

This research is supported primarily by a grant from the National Science Foundation, NSF–MDR–88–50363, to Andrea A. diSessa. I also gratefully acknowledge a contribution to support students from Apple Corporation, and a contribution of equipment from Sun Microsystems.

Correspondence and requests for reprints should be sent to Michael Leonard, Education in Mathematics, Science, and Technology, Graduate School of Education, University of California at Berkeley, Berkeley, CA 94720.

to hit "blobs" on a coordinate grid. Although my program had many rough edges, I was impressed by what I had been able to accomplish in a day in Boxer, as compared with what I could do in that amount of time with other languages such as Logo, Lisp, or Basic.

Shortly thereafter, I was presented with the opportunity to work with a group of seventh- and eighth-grade advanced math students at Bentley School in Oakland, California. These 4 seventh- and 4 eighth-grade students had been working with Boxer approximately one class period per week, since the start of the school year. For the first month and a half, I acted as a tutor, helping these students to work on various programming assignments related to their math class. One of the topics they had explored in class was fractals. They seemed to be very interested in fractals, and their teacher was eager to have them learn how to program fractal designs in Boxer. I decided to try to teach them recursive programming in Boxer so that they could program fractals. I also wanted to study their learning of recursive programming as a research project.

Recursive programming seemed like a particularly good area in which to study the students' learning of Boxer. Recursion is recognized to be a difficult topic in computer programming, for adults as well as children. Children have been found to have significant misconceptions related to recursive programming (Kurland & Pea, 1989). I thought it likely that the students might develop one of several misconceptions identified by Kurland and Pea (1989, p. 300), specifically, the "mental model of embedded recursion as looping." I also thought that Boxer's visual structure might be particularly effective for helping them to overcome this misconception.[1] Finally, the students' familiarity with fractals from math class and their interest in exploring them on the computer seemed compatible with one of the main goals of Boxer, which is to enable students to use computer programming to explore significant topics in mathematics and science.

I developed a curriculum in Boxer for teaching the structure of recursive programs. I did not attempt to teach recursive programming strategies, although I will attempt to do so in future versions of the curriculum. My curriculum had four parts: (1) simple tail-recursive turtle graphics programs that continued to run indefinitely; (2) programs similar to those in part 1, but including input variables; (3a) embedded recursive programs with a single recursive call; and (3b) embedded recursive programs with multiple recursive calls. Each part contained an introduction, example programs, "challenges," and a summary. The curriculum asked the students to explore the examples in various ways, to find out what they

[1]Boxer was designed to have a clean and complete structural core based on the *box* as the fundamental computational object. In Boxer, the "box" (which may be displayed as an area enclosed by a rectangle on the screen) is used to represent both procedures (do-it boxes) and data (data boxes). Procedures and data may be referred to by name if their boxes are given names. Hierarchical program and data structures may be created by placing boxes inside of boxes. The amount of information displayed on the screen may be precisely controlling by "opening" and "closing" boxes. For text-processing and screen arrangement purposes, boxes are treated like characters of text.

did, how they could be modified, and (especially) how they worked. The curriculum provided a model of the operation of recursive programs that was designed to help students to overcome the "recursion as looping" misconception. The challenges were programming problems that asked students to apply what they had learned in order to write recursive programs to create particular designs. In Section 3b, the examples and challenges involved fractal designs. An outline of the full curriculum is shown in Figure 1.

This study focuses on the learning of 1 eighth-grade girl (S), who worked with a partner (P), on the curriculum. Figure 1 shows, in regular type, the portion of the curriculum that they completed. (They returned voluntarily two months later to finish the curriculum during the summer. Their summer work is currently being analyzed.) I believe that S's learning about recursion was confined to places where impasses arose when she was attempting to solve programming problems. Impasses for her arose on the squiral program[2] in part 2, and on the count-b program[3] in part 3a.

This article is organized as follows. Sections 2 and 3 provide background for the analysis. Section 2 discusses impasse resolution, the analytic frame for this article. Section 3 sets forth the structure of recursive programs in Boxer: the structure that students were intended to learn from the curriculum. Section 4 explains the method, including impasse-analysis of videotape data. Section 5 shows what S knew about the structure of recursive programs during three periods: before the squiral impasse, between the impasses, and after the count-b impasse. Section 6 discusses a number of issues including generality of the results, role of the curriculum, modelling the learning of structural knowledge, evaluating Boxer, and teaching programming. Section 7 presents conclusions.

2. LEARNING VIA IMPASSE RESOLUTION

Impasse resolution has been proposed by Newell (1990; Laird, Rosenbloom, & Newell, 1986) as a learning mechanism that might account for all learning. "Impasse" is defined as a failure to be able to proceed toward an established problem-solving goal ("getting stuck"). "Impasse resolution" is the process of getting around the difficulty ("getting unstuck"). Newell focuses on the implementation of impasse resolution in a general artificial intelligence problem-solving program, Soar. Although I believe the formulation of impasse resolution used in this article to be compatible with Soar's, my formulation does not make a commitment to any particular implementation of impasse resolution, or to other assumptions made in the Soar theory.

VanLehn (1988) proposed a theory of impasse-driven learning that might

[2]Squiral is an exemplar recursive program that draws a "square spiral."

[3]Count-b is an exemplar recursive program that counts backwards in a way that is initially surprising to students.

Introduction
Part 1: Simple tail recursion
    Introduction
    Examples
        spinsquare program (subsequent examples follow a similar format)
            discussion/model
            exploration
        movesquare program
    Challenges
        "sunburst" design
        "starcircle" design
        make up your own challenge
    Summary
Part 2: Tail recursion, with input variables
    Introduction
    Examples
        count program
        growsquare program
        squiral program (first impasse)
    Challenges
        "target" design
        "butterfly" design *(partially completed)*
        "spiral" design
        make up your own challenge
    Summary

Part 3: Embedded recursion
- Introduction
- Part 3a: Singly-embedded recursion
  - Introduction
  - Examples
    - count program
    - **count-b program (second impasse)**
    - growsquare1 and growsquare2 programs
    - spinsquare1 and spinsquare2 programs
    - summary
  - Challenges
    - "sweeper1" and "sweeper2" designs
    - "rects" design *(only partially completed during this study. The rest of the curriculum was completed in a later study)*
  - *Summary*
- *Part 3b: Multiply-embedded recursion*
  - *Introduction*
  - *Examples*
    - *snowflake program (Koch snowflake curve)*
    - *cornflake program (a variation on snowflake)*
  - *Challenge*
    - *"Sierpinsky's carpet" design*
- *Summary*

Figure 1. An outline of the Boxer curriculum for teaching recursive programming. Programs where an impasse arose are in boldface type. Parts of the curriculum that were not completed during this study are in italic type.

account for the learning of procedural skills. His theory focused not only on the learning of correct knowledge, but also on the generation and remediation of incorrect knowledge (bugs). He explained a number of important aspects of impasse-directed learning which I believe can also be seen in the results of this article.

> [With impasse-directed learning,] learning only occurs when the current knowledge base is insufficient. Moreover, it is not just any incompleteness that causes learning. The incompleteness must be relevant enough to the person's affairs that it actually generates an impasse. The person's problem solving must require a piece of knowledge that is not there. Consequently, one learns only when there is a need to learn (VanLehn, 1988, p. 37).

Thus, learning via impasse resolution only occurs when knowledge is insufficient to resolve an impasse. However, there is no guarantee that correct knowledge will be acquired. In fact, one of the strengths of the theory is that it can be used to explain both the acquisition of wrong knowledge (misconceptions) and remediation of such wrong learning.

The following is a hypothetical scenario of learning via impasse resolution. Although this scenario is much simplified from any actual learning episode, it is substantially consistent with the results of this study. I also believe that this basic impasse-resolution learning process can be generalized.

Figure 2 illustrates the scenario. A student enters with an initial knowledge state and a goal. The student engages in some problem-solving activity to move towards the goal. An impasse arises, which blocks the student from moving towards the goal. The student sets up a subgoal to resolve the impasse. Because the student does not yet know exactly what is causing the impasse, the student searches to localize the problem sufficiently, setting up specific subgoals, which, if achieved, would resolve the impasse. If one of these subgoals is achieved, the impasse is resolved, and the student can once again proceed towards the original goal. Learning happens when the subgoal is achieved and the impasse is resolved. The learning consists of a new knowledge element which takes into account the particular aspects of the situation that caused the impasse, and

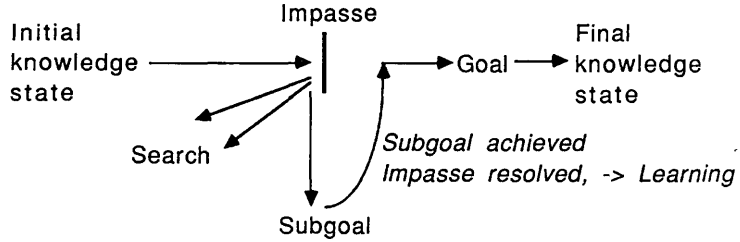


Figure 2. Learning via impasse resolution.

combines them with new knowledge discovered by achieving the subgoal. In similar future situations, the learner may recognize conditions that formerly resulted in an impasse. If this happens, the new knowledge gained earlier is retrieved, and the impasse is avoided.

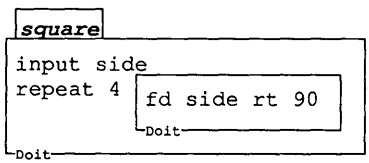
Impasse resolution is a learning process that produces knowledge elements within particular problem-solving contexts. I do not take a stand here on how these knowledge elements are integrated to form broader changes in knowledge structure, or on whether other mechanisms are needed to explain learning of other types of knowledge. The methods I have developed for impasse analysis as an empirical technique are explained in detail in Section 5.

3. THE STRUCTURE OF RECURSIVE PROGRAMS IN BOXER

Recursion in programming can be defined as a situation where a procedure calls *itself*, either directly or indirectly. Therefore, the way that recursive programs work is determined by the way that procedure calls work. The structure of Boxer is designed to be concrete and easily understandable to students. However, some aspects of Boxer's operation are hidden when procedures are running normally, because the user does not always need to know what is happening internally. The internal processing of procedures can be made visible in Boxer by showing what is happening using a model. Unlike most other languages, Boxer can model its own execution quite effectively and concretely, using the basic structure (a box) built into the language.

The following narrative description of procedure calls in Boxer applies to all procedure calls, including recursive ones. It is intended that students understand procedure calls in this way, and that they will be able to apply this knowledge to understand how programs work. In Figures 3 through 6, the narrative is illustrated using a simple nonrecursive procedure "square" which draws a square of any size.

When Boxer is executing and it comes to a name, it looks up that name. If it finds that the name refers to a do-it box (a procedure), it makes a temporary copy of the do-it box and executes the copy in place of the name. If the first line of the




```
==> square 100
```

Figure 3. "Square" is about to be executed with an input of "100" ("==>" points to the next thing to be executed; "==>" does not appear in Boxer).

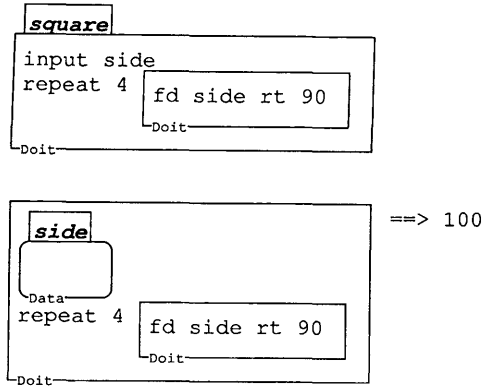


Figure 4. Boxer found that "square" refers to a do-it box, so it created a temporary copy of the do-it box in the place of the name. Because the first line started with "input," a data box was set up corresponding to the word "side." Boxer is about to seek a value for "side."

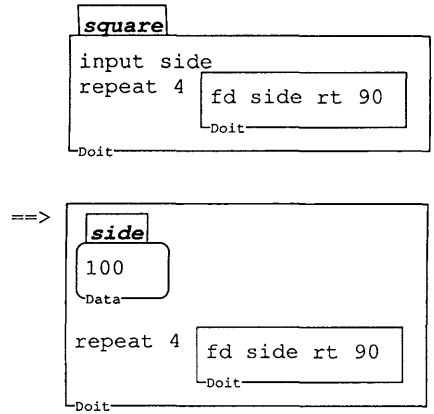
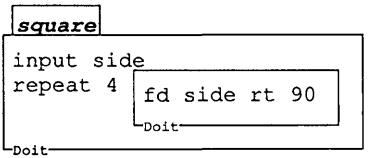


Figure 5. Boxer found "100" and installed it as the value of "side." Boxer is about to execute the copy of "square."

do-it box starts with "input," Boxer sets up a data box (variable) in the do-it box copy corresponding to each word after "input" on the first line. For each input data box, Boxer seeks a value, which can be found in order on the line immediately after the copy. Each input data box is assigned a value. The lines inside the copy are then executed. When the copy is done executing, the copy is thrown away. If a value was returned by the copy, it is available for further processing.

The Figure captions do not completely describe the example. Additional structural features that govern Boxer's behavior in this example are: (1) When a do-it box is executed, control flows through the box to the right and down, like reading English text; (2) The "fd" procedure takes one input, and moves the




```
square 100  ==>
```

Figure 6. The repeat command inside "square" was executed, drawing a square. (The square would have been drawn in a graphics box, which is not shown here.) After that the temporary copy was thrown away. Control was returned to the original line, just after "square 100." Because there was nothing else on the line, execution stopped.

turtle forward by that amount, drawing a line if the pen is down; (3) The "rt" procedure takes one input, and turns the turtle that number of degrees to the right; (4) The repeat procedure takes two inputs, a "number" and a do-it box. It causes the do-it box to be executed "number" times; (5) When Boxer encounters names like "repeat," "fd," and "rt," it executes them exactly as it executes "square." The only difference is that repeat, fd, and rt are "primitive" procedures built into Boxer.

This comprehensive view of procedure calls is called *structural* knowledge because it addresses the structure of the programming language. The structure of a programming language determines how programs written in the language will be executed, as well as how data may be structured. The structure includes such things as syntax, flow of control, and scoping. All procedural programming languages (such as Logo and Boxer) share many common structures. Each language also has its own specialized structures or variations on structures.

During programming, many different types of knowledge may be accessed and used. Various types of programming knowledge are briefly illustrated here with regard to the "square" procedure example. Structural knowledge of square might remain in the background. It might be thought of in purely *functional* terms (e.g., "It makes a square of any size whenever I need it.)", as a *template* (e.g., "I need to make a 'triangle' procedure, let's see, a triangle is quite similar to a square, I'll just take the square procedure and modify it to make a triangle."), as an element in a *plan* ("To make this design of a house, I'll use a large square, with a group of smaller squares for a window, and a rectangular door . . ."), as an element in a *strategy* ("Should I attack this programming problem from the top-down or the bottom-up? I think I'll proceed from the bottom-up. First I'll write procedures to make the elementary shapes such as squares and circles, then I'll work on putting them together into a larger design . . ."), or as part of a *belief* ("I can write this program quickly, because it's made up of easy pieces like square which I have used many times before."). All of these types of knowledge are important for complete programming skill. This study focuses on understanding of the structure of recursion only.

4. METHOD

An eighth-grade girl (S) was videotaped as she worked with another eighth-grade girl (P) over six sessions. Total time was approximately 3 hours. Both girls had been using Boxer for about one school year, one 40-minute session per week. They had completed a number of turtle graphics programming projects, but had not used recursion before the study. For this study, the pair used a curriculum written in Boxer that was designed to teach a correct structural model of recursion. I have analyzed S's evolving knowledge of recursive structure in a fine-grained way. The analysis focuses on understanding two impasses that arose and how they were resolved. The first impasse deals with understanding inputs and parameters for procedures. The second deals with understanding embedded recursion.

Impasse Analysis

This section explains how impasses are analyzed in this study. Here, an impasse is defined as a failure to be able to proceed in problem solving, recognized by the learner.

The goal of impasse analysis is to weave a rich tale, centered on an impasse, that delineates as completely as possible the content and process of learning. This analytic process is summarized below in the form of questions to be answered by the analyst. The questions are ordered roughly in the order in which they would be answered and data would be presented in support of the answers, not in the order that the analysis would be carried out.

- Roughly, what is the impasse?
- What is the student's initial knowledge state?
- What is the student's goal?
- Does the student recognize an impasse?
- How does the student search to localize the source of difficulty?
- What subgoal does the student set up to resolve the impasse?
- How does the student achieve the subgoal?
- What was learned by the student?
- What is the student's final knowledge state?

The notion of "weaving a rich tale" is taken seriously in this analytic process. The process does not proceed step-by-step from roughly identifying an impasse to showing what was learned. Instead, the process is one of gradually collecting evidence to support a more and more detailed description of the impasse that answers all of the questions. Part of the process includes consideration of alternative explanations for each item. For example, alternatives to the view that this is indeed an impasse for the student must be considered. If it is not an impasse for the student, then any further analysis using this model crumbles. Fortunately, the

items in the checklist tend to reinforce each other, so establishing one item tends to help establish others. On the other hand, failure to be able to establish one item can seriously threaten the rest of the analysis. For example, the student should learn something by resolving an impasse. Therefore, there should be strong evidence of a long-term change in knowledge. If there is not, then the notion that this was an impasse for the student must be questioned, or the nature of the resolution should be reconsidered. (Maybe the student got around the impasse by some means other than learning what the experimenter or teacher intended the student to learn.)[4]

The primary tool used for identifying and analyzing impasses is the transcript. The transcript gives the experimenter flexibility to focus on many data points from separate times simultaneously or in quick succession. *All* lines of the transcript which relate to a given analytic question must be examined. In this study, transcripts were made from videotapes. The videotapes were referred to, as needed, to clarify ambiguities in the transcript or to get the flavor of the interaction that was lost in the transcription process. Now I will briefly describe how each of the impasse-analysis questions can be answered using the data.

Roughly, What is the Impasse? An impasse can usually be identified roughly because the student is doing something, experiences difficulty, says something which makes it quite obvious that the difficulty is recognized, and proceeds in a different direction to resolve the difficulty. The two impasses that form the heart of this article's results were easy to identify just by watching the videotape. In the squiral impasse, S was trying to trace (mentally run) the program. This was something that she frequently did while analyzing examples or debugging her work on challenges. On squiral, she could not do it. Eventually she asked a question about how the program worked, and received an answer that allowed her to trace it. In the count-b impasse, S made a prediction by tracing the program. She tried the program and saw that the prediction was wrong. This led to an impasse on her goal of correctly tracing the program. She set out to understand why she was wrong, and finally did understand with help from a model and the experimenter.

What is the Student's Initial Knowledge State? Understanding the student's knowledge state before the impasse is a difficult task. It involves looking at successful and unsuccessful problem solving by the student, before and after the impasse, to determine what the student could and could not do. This might involve looking at a large amount of data. For example, to determine S's knowledge state before the squiral impasse, I looked closely at her work on nine other

[4]Of course it is possible that impasse analysis might fail because it is not an appropriate method for attacking some particular data. I am currently assessing the breadth of applicability of the method by attempting to apply it to analyze learning of other types of programming knowledge.

programs in about two thirds of the transcript. Then I had to induce from *all* of these activities before, during, and after the impasse *two* knowledge states—one before the impasse, one after—that fit with all known data. It is impossible to know everything that the student was thinking, therefore, it is important to specify what can be stated about the student's knowledge and backed up with reasonable evidence, and what cannot be said for sure. How could data obtained after the impasse serve to clarify the student's knowledge state before the impasse? This is simple: I presume that only one knowledge element is created by resolving an impasse. Data obtained afterward is just as relevant as data obtained before for clarifying the state of other knowledge elements.

What is the Student's Goal? Establishing the student's goal can be difficult because it means knowing not only what the student is doing, but also why the student is doing it (from her point of view). In the squiral impasse, S was acting as if she were trying to trace the program but never explicitly said this. In order to rule out some other interpretation of her actions, I had to look over a large portion of the entire transcript to see that her activity on this program was part of a larger pattern where she and her partner traced programs and talked about what their output *would be,* without actually running them. In the count-b impasse, her goal was obvious because she readily made a prediction, and her actions and statements indicated that she was determined to discover why it was wrong (i.e., her goal was not just to make a prediction, but to make a *correct* one).

Does the Student Recognize an Impasse? It is easy to see when a student's activity has a break in it. However, it is difficult to know when the break forms an impasse for the student, in other words, that the break is preventing direct movement towards a goal. For example, S frequently leaves out parameters after procedure calls, and receives error messages from the computer indicating that the program won't run and why. These errors never form impasses for her. She just puts in a parameter and goes on. When she stops trying to run squiral and instead asks, "what does that line (the recursive call) really mean?" she is clearly recognizing an impasse and setting up a subgoal, to understand the line. When she gets her prediction for count-b wrong and sets off to find out why, she explicitly recognizes that she has predicted wrongly and sets up a subgoal to find out what was wrong with her understanding.

How Does the Student Search to Localize the Source of Difficulty? In each of the impasses in this study, S acted purposefully between the time when she was first blocked from achieving her goal (whether she explicitly recognized the block yet or not) and when she set up the specific subgoal that resolved the impasse. During this time she seemed to be setting up subgoals to rule out various possibilities that could have caused the impasse, but had not. This process appears to have been a central part of how she overcame the impasse. In

general, the description of the search involves describing the subject's actions from her first difficulty to her formulation of a subgoal that would resolve the impasse, along with discussion of what purpose these actions seemed to have for her.

What Subgoal Does the Student Set Up to Resolve the Impasse? The analyst is trying to determine what subgoal is set up and achieved that finally resolves the impasse. In squiral, this was easy because S's question, "What does that line really mean?" is clearly setting up a subgoal to gain new understanding. In count-b, the situation is a bit more complex, because she had two pieces of existing knowledge that led her to make a wrong prediction. Her subgoals were not asked as explicit questions, but could be paraphrased loosely as, "my knowledge says that something additional should have to be in this program in order for it to produce the output I have observed. Nothing additional is in the program, so what is wrong with my knowledge?"

How Does the Student Achieve the Subgoal? This is central to the analysis. In order to analyze the impasse effectively, it is necessary to explain how the student could use her initial knowledge, together with something in the situation, to achieve the subgoal and resolve the impasse. The result of resolving the impasse is a new piece of knowledge. Therefore, there must be a plausible match between how the subgoal was achieved and the difference between the initial and final knowledge state. As an example, S brought a useful piece of knowledge into the squiral impasse: how to change the value of a variable. Her partner provided useful information: how to match parameters with inputs. Together, these two pieces of knowledge allowed S to achieve her subgoal, to understand the relationship between two lines of the program. This local process corresponded with the global change from initial to final knowledge state, the addition of a new piece of knowledge, which I call "match and change."

What was Learned by the Student? A new piece of knowledge should have come into being as a result of resolving the impasse. If the preceding questions have been answered, this one is easy. What is learned is precisely the difference between the initial and final knowledge state. When I write up the analysis, I give this new piece of knowledge a name for easy identification and explain how it was constructed out of preexisting knowledge and something in the situation.

What is the Student's Final Knowledge State? This is easier than specifying the initial knowledge state, because much work has already been done. The main focus is to establish exactly what is different from the initial knowledge state by looking at students' work during and after the impasse. Any learning via resolving the impasse must be confirmed to exist by performance after the impasse. In addition, the limits of the learning must be specified by looking at where the

student still has difficulty after the impasse. For example, match and change from the squiral impasse was demonstrated to exist on a number of subsequent programs, and its limits were clearly delineated on the "target" design challenge and on the count-b impasse.

Some comments about the standards of evidence are in order. Facts are gathered, alternative explanations are weighed, and finally, *all* of the relevant evidence must support whatever conclusions are reached in the analysis. If some piece of evidence does not fit, either the conclusions must be modified, or some explanation must be found for the evidence that shows how it is plausibly not a contradiction after all.

However, I take impasse analysis to include an essential social process. After preliminary conclusions are developed, the next step is to bring the conclusions to an outside audience. I brought preliminary conclusions to several individuals and tested and refined the conclusions in the light of comments and criticisms. The following test was to present and defend the entire set of results in a group setting, in this case, that of Schoenfeld's "functions" research group at U.C. Berkeley. Finally, I wrote an early version of the results section to try to answer criticisms raised. Another round of defense in front of the functions group ensued, and additional feedback was obtained from individuals. Impasse analysis requires understanding not only of the local problem-solving episode, but also of a broad context of problem solving surrounding the episode. For this reason, argumentation over the conclusions in front of increasingly wider audiences seems to be the best way of ensuring the reliability and validity of the analysis.

5. RESULTS

The results will show S's knowledge state at three points in time: before the squiral impasse, between the impasses, and after the count-b impasse. The analysis shows (1) that it is possible (though difficult) to chart knowledge states and changes that localize learning to particular events with particular results; and (2) that the form of learning is compatible with the impasse-resolution scenario and that learning is limited to that sufficient to overcome the impasse. At certain points, detailed analysis is suppressed in order to go on with the main point of how the impasses arose and were resolved.

Initial Knowledge State

This section describes aspects of S's initial knowledge state before she encountered her first impasse. Of course, this is only a small portion of her actual knowledge state (the portion most relevant to this study). An example of her work during this knowledge state is given after the description of the state.

S had quite a bit of correct knowledge about Boxer's structure. All of her knowledge of how to navigate in the Boxer environment was correct (at least so far as it was required for this study). Likewise, all of her relevant knowledge for

accessing and changing variable values was correct. Part of her knowledge of control was correct, including simple control (to the right and down through a procedure, like reading English text), special control for the "repeat" command, and special control for the "if" command. Her knowledge was flawed in four ways, all with respect to procedure calls.

First, she had a "naive interpretation" of how to change the value of input variables, which can be described as follows: S is able to interpret the meaning of some parameters in procedure calls. If a parameter is a number, the corresponding input variable takes on the value of that number. If a parameter involves a variable in a simple way, the "obvious" thing is done to the variable's value. For example, a call to the "growsquare" procedure, "growsquare n+4," means add 4 to the value of *n*, and then jump to the start of growsquare and execute from there. In this view the effect of the procedure call can be interpreted without concern for its relationship to the input line of the procedure. This type of misconception was referred to in a general way by Kurland and Pea (1989, p. 300) as "decontextualized interpretation of commands." Without understanding the relationship between the procedure call and the inputs to the procedure, a procedure call with two variables combined in one parameter (such as "growsquare n+i") could not be interpreted unambiguously by S. Her naive interpretation led to the squiral impasse in exactly this way.

Second, she had a "jump and execute" model of procedure calls. In other words, she thought that a procedure call caused control to jump to the start of a procedure and execute from there. In the case of recursion, this concept applied repeatedly leads to the "recursion as looping" misconception described by Kurland and Pea (1989). This misconception led to the count-b impasse.

Third, S thought that parameters to the procedure were optional, and continued to think this at the end of the study. In other words, she did not realize that if a procedure contains an input line, then each time the procedure is called, values for its input variables *must* be found. This is another variation of Kurland and Pea's "decontextualized interpretation of commands." Although this "optional parameters" misconception is strongly related to naive interpretation, it did not lead to any impasses during this study. As far as I can tell, S still believed that parameters were optional to the end.

Finally, S had a misconception that I have labelled "two contexts." She viewed procedure calls executed directly by the user as a separate structure from procedure calls executed indirectly from within a procedure, such as a recursive call. Although she received feedback in her programming several times, which the experimenter believed could have led her to address this misconception, it never led to an impasse, and her misconception apparently did not change in the course of the study. Figure 7 summarizes S's initial knowledge state. I am suppressing detailed analysis to establish S's initial knowledge state at this point, so that we can go on with the main point of the impasses and how they arose.

S did not think that her understanding was faulty. It had served her quite well

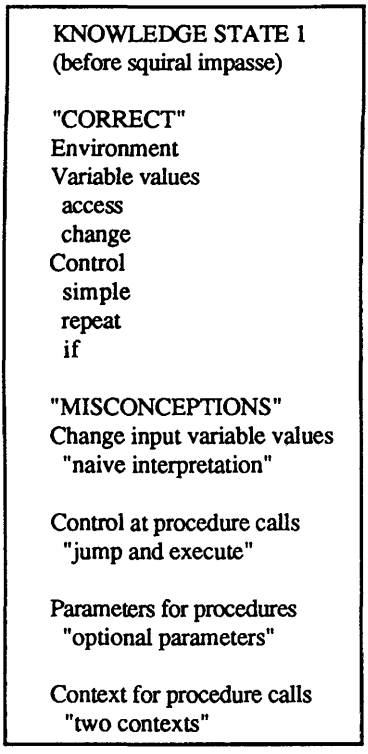


Figure 7. Relevant portion of initial knowledge state, before the squiral impasse.

up to this point. Based on it, she thought she understood many procedures before growsquare, and was able to see how growsquare produced its output. Growsquare takes a length of a side as input, and draws a series of squares having the same lower left corner, with the smallest square having side equal to the first input, and subsequent squares being larger by four turtle steps. Figure 8 shows S's first successful execution of growsquare, with an input of 50. The program would have kept going indefinitely, but was stopped here before the turtle went off the edge of the graphics box.

Figure 9 is a model provided to the students of how growsquare works with a starting input of 50. A copy of growsquare is created with a copy of side equal to 50. Then "repeat" executes to create a square. Finally, the recursive call causes another copy of growsquare to be created. A new copy of side now has value 54. Another square (size 54) is created. This process of creating copies continues indefinitely to draw the design. The original values of side are not changed. Only the copies of side have new values. Execution takes place by making copies of growsquare, not by jumping back to the beginning of growsquare. Although S

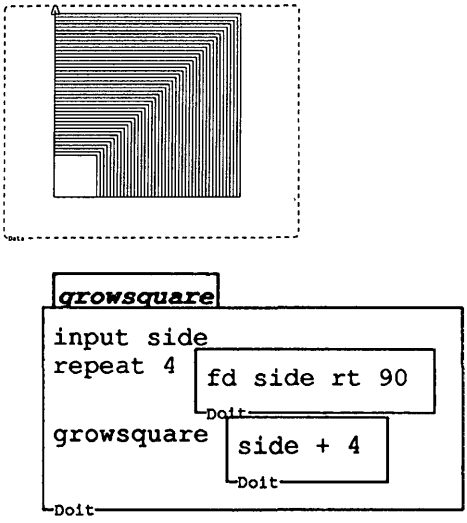


Figure 8. Growsquare program and output.

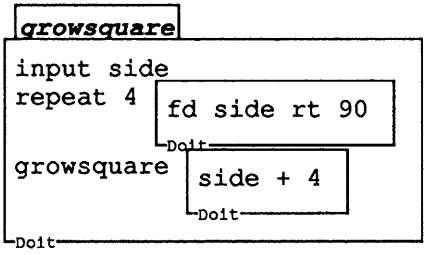
looked at this model and thought she understood the program, she did not have the understanding shown by the model and described in this paragraph.

Figure 7 showed how S understood programs such as growsquare. According to her naive interpretation the recursive call "growsquare (side+4)" means "add 4 to side." According to jump and execute, control would then jump to the start of the procedure, and execute from there. Optional parameters was revealed when she first tried to execute growsquare, with no inputs. Two contexts was suggested by the fact that in the recursive call, growsquare clearly takes one input, "side+4." In spite of this, she first tried to execute growsquare directly with no inputs.

Notice that, although S's understanding is incorrect, it does allow her to trace the program successfully. She did this, and filled in exactly the correct numbers for the input values in the model of the program. This understanding broke down on the next program, resulting in the first impasse. For the first impasse, I will present enough of the discussion and the protocol data to give the reader a fairly good sense of how the analysis was done. For the second impasse, I will be more telegraphic.

First Impasse

Figure 10 shows the squiral program and the output generated by "squiral 10 20," the procedure call that S and P used to try out the program. S's optional




```
growsquare 50
```

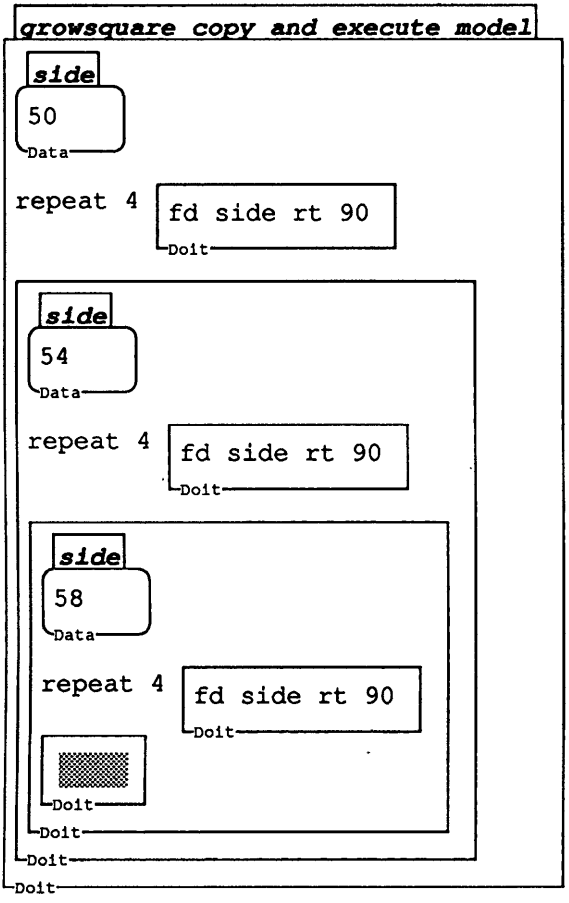
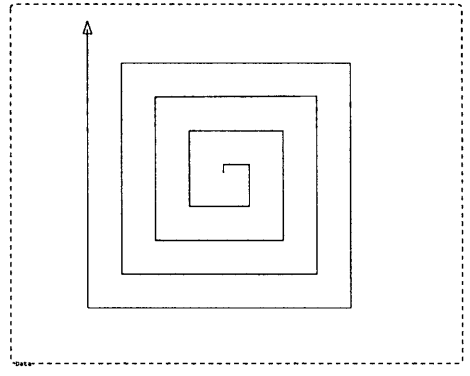


Figure 9. Copy-and-execute model of the growsquare program.



```
squiral
input side inc
fd side
rt 90
squiral  side + inc  inc
         Doit
Doit
```

```
squiral 10 20
```

Figure 10. Squiral program and output.

parameters misunderstanding was revealed when she first tried to execute the program using just "squiral" (with no parameters). Then she tried "squiral 10," and finally got it to work using "squiral 10 20."

The following dialog suggests that her goal is to trace the program in order to understand how it produced its output. She had done this for other exemplar programs without difficulty. She apparently recognizes that she has an impasse. The last line of dialog shows how she begins the search to localize the impasse. (My commentary is in italic type, and appears after the line to which it refers. The punctuation of the dialog is sometimes nonstandard. This is an attempt to capture more of the pace and tone of the dialog.)

S OK. So it goes forward side, right 90, and . . .
S is stepping through the program line by line, and pauses when she reaches the last line, the recursive call.
P Side plus, and then it goes . . . . . [long pause]
P describes part of the recursive call, and also pauses.
S Let's look at the model.

S is apparently comfortable stepping through the program until the recursive call. (The problem has to do with S's optional parameters and naive interpretation

misconceptions, but S does not yet know the precise nature of the problem.) Then she suggests looking at a copy-and-execute model of squiral (like the one provided for growsquare). Apparently, she hopes to localize her source of difficulty by looking at the model.

S decided to hand simulate the program instead of looking at the model. This is shown in the following dialog. I don't have a good explanation for why she made this decision, except that she thought it might help her to localize her source of difficulty, by allowing her to trace the program one line at a time. Figure 11 shows the screen with output as it appeared *after* the following dialog. The line at the bottom of the Figure is what she typed ("fd 10 rt 90 fd 30 rt 90 fd 50"). The graphics box shows the output. The lines in boldface type are critical, and will be discussed later.

P OK you have these two things side and inc, I don't know, increment. Increment? [She points the inputs out with the mouse.]
P starts out trying to explain, but ends up sounding confused herself.

S Wait, will you put me down there. [She points to space below graphics box.]
S asks P to place the cursor below the graphics box so that she can hand run the program line by line, with numeric values substituted for variables.

P [She puts the cursor down there below the graphics box.]

S I don't know. [inaudible] Let's try it.

P Forward 10.

S [She types and executes "fd 10." The turtle starts at the middle of the screen and makes a short line going up, the first part of the "squiral 10 20" design.]

P Right 90.

S [She types and executes "rt 90." The turtle turns and faces to the right.]

P OK and then you go ffffff . . .
P told S to type fd 10 and rt 90. Now it appears that she is trying to allow S to proceed on her own. Her "fffff . . ." seems to be there in order to suggest that S give the next command.

S And then I go, forward 10, right
S has said forward 10, and is about to say right 90.

P Wait don't you go, forward, um, 30 [pointing to the recursive call] Because we made inc, so you go forward 30.
P corrects S's mistake, gently.

S [She erases fd 10, types fd 30.]

P Right 90, fd 50, rt 90.
Now P is not allowing any further chances for mistake.

S [She types what P has said, then executes the whole line, "fd 30 rt 90 fd 50 rt 90."—it makes the start of the design that "squiral 10 20" produced.] Yeah. [She erases the experiments below the graphics box.] So it does this little recursive thing [inaudible].
S sees that the sequence of commands provided by P is beginning to produce the same design that "squiral 10 20" produced. Her "yeah" indicates that she recognizes this. Her description of what the program does is pretty vague: "it does this little recursive thing."

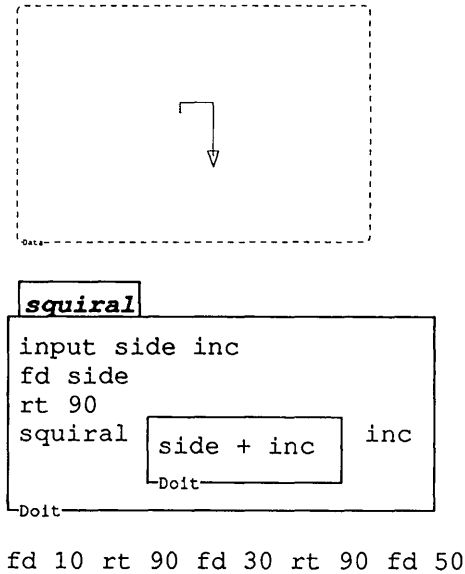


Figure 11. Hand simulation of squiral program.

P Yeah. . . . Oh I think that will help me on my project.
P understands the program, and thinks she can use something from it on another programming project that she is working on.
S OK let's look at the model.
Apparently, S is once again asking for help.

The lines in boldface show how P tried to let S fill in the amount to go forward after the first recursive call. S gave the wrong number, 10. I believe that this was just a guess, and that S needed to guess because her naive interpretation failed on this multiple input case. P then corrected her. After this, it is obvious that P understands the recursive call, and that S does not. Now it seems clear that S's goal is to trace the program to produce its output, that she recognizes that she has an impasse, and that she has localized the impasse to the recursive call. However, she still is not sure what is wrong with her understanding of the recursive call. In the last line of the dialog, S continues her search by seeking more help from the model.

When S and P open the model of squiral, they find that a finished model of squiral is not given to them, as it was with growsquare. Instead, they have to build the model for themselves. In the dialog that follows, we see that P does not know how to build a model. S does know how, but does not understand how the program works well enough to build a corresponding model. Eventually S is led to pose a very specific question about the program. By formulating this question,

S has localized the impasse sufficiently (although she doesn't yet know this until after the question is answered). She has, in fact, set the subgoal (to get an answer to her question), which will resolve the impasse when it is achieved.

P OK what does it mean build a model of squiral starting with a side of zero and an increment of 2?

Apparently P doesn't understand what it means to build a model. This is not surprising since S has built and/or completed all of the models up to this point. P's statement could be taken to mean that she doesn't understand what squiral would do starting with side zero and inc 2, but her subsequent explanation rules out this alternative explanation. P does not know how to build a model, but does understand the program (as shown in the next segment of dialog). S knows how to build a model but does not understand the program. Thus, S and P must cooperate in this situation.

S I don't know. I'm not sure but we're supposed to do one of these model thing, but if it has an input, then won't it have to read in thing [she gestures a "continuing" motion]. each time?

S seems to be trying to build a model (like the growsquare model) mentally, and asks a question which is difficult for me to interpret with certainty, and is apparently difficult for P to interpret as well. S is probably thinking of inputs as getting a value that the user typed.

P What here? [She points to parameters in recursive call, first the box containing "side+inc" and then moving across both parameters "side+inc" and "inc."] No that's

P seeks clarification of S's question.

S **If it's input, well what does that line really mean? [points to the recursive call]**
And finally, S asks a really clear question.

In the boldface line above, S asked a question that would finally allow her to overcome the impasse. She refers to the first line of the program ("If it's input") and points to the last line of the program as she finishes her question ("what does that line really mean?"). Although she doesn't know it yet, she has successfully localized her difficulty: the relationship between input and the parameters in the recursive call. She has set a subgoal (to find out what this relationship is) that will be achieved, and will resolve the impasse. This may be the first time that S has sought to understand the relationship *between* two lines. She is directly addressing one of her misconceptions that was called "decontextualized interpretation of commands" by Kurland and Pea (1989).[5] She is moving beyond trying to understand these program lines in isolation, and is seeking to understand the relationship between them.

In the dialog below, P answers S's question. Her answer shows S how to use a specific procedure that is an improvement upon her naive interpretation of the

[5]Of course she isn't addressing all aspects of this misconception, but rather only the one that she has just become aware of: that she should know the relationship between these two lines.

recursive call. S achieves the subgoal that was set above, and she comes to understand the relationship between the two lines.

P OK squiral requires two inputs, right? [She points to recursive call "squiral," the last line of the program, to "input side inc," the first line of the program, back to recursive call first parameter "side+inc."] This is the first input.
With her finger movements, P has established that she is talking about the relationship between the input line and the recursive procedure call. She has focused on the first part of this relationship, "the first input."
S Right.
P This is the second input [moves finger to second parameter "inc"] Right?
S Oh I see . . . I see I see
P So if this [pointing to the first variable in the first line of the program, "side"] is 10, and this [pointing to the second variable "inc"] is 20, then this [pointing to parts of the parameter "side+inc" in the recursive call] into this added . . .
S So it's saying change side to side plus inc,
P Right
S And to keep inc the same.
P Right. Exactly. [nods]
S So you increase it [side] by inc each time [makes a progressive chopping motion]
P Right. That's why it's called, INCrement.
S OK I think I've got it now.

In the first five lines of this dialog, P is answering S's question. P acts out and explains an explicit matching procedure that shows how each parameter corresponds to a particular input variable. S responds briefly with "Right," and "Oh I see . . . I see I see." In the last seven lines, S is explaining her new understanding to P, in order to get confirmation. S relies on her prior knowledge (how to change the value of a variable) for her explanation.

Taken together, these dialogs, along with the rest of the data, appear to provide reasonable confirmation for the impasse-resolution scenario. It appears that an impasse existed, was resolved, and that S constructed a new piece of knowledge. However, a simpler explanation should be considered if one is available. One simpler explanation might go something like this:

> S and P have muddled along until P told S (some part of) what she did not know. Now S knows the thing that P told her.

Unfortunately, this explanation does not fit with data outside the scope of that presented in detail in this article. The whole body of data contains a number of instances where S read something, was told something, or received feedback that could have told her something, without any apparent long-term effect on her knowledge. S seemed to be ready to learn something about programming when

the new knowledge would resolve an impasse for her, but otherwise was not yet ready to learn.

Roughly, What is the Impasse? She wanted to trace the program, and was unable to proceed, probably due to some difficulty understanding the recursive call.

What is the Student's Initial Knowledge State? Her initial knowledge state was discussed earlier and was summarized in Figure 7 (see also Figure 12). The part of her knowledge state most relevant to this impasse was naive interpretation. She was unable to use it to interpret the recursive call. What sense could S make of the recursive call in squiral using her prior knowledge, naive interpretation? Using her naive interpretation, the analysis would be something like this:[6]

> Side and inc are added. So, something should be increased. Should side be increased by inc, or should inc be increased by side? Then there is a second parameter, inc alone. So nothing is done to inc. If nothing is done to inc, then why is it there at all? Maybe inc is not a parameter, but instead is a command that has some unknown function.

Remember that for S, parameters are optional. Therefore, there is no way to make sense of "inc" at the end of the line. If a parameter is not required then why provide one just to maintain the same value? She is ready to learn something because she has a goal, and she cannot move towards it based on her prior knowledge alone.

What is the Student's Goal? She wanted to trace the program to see how it produced its output.

Does the Student Recognize an Impasse? Yes. There is suggestive evidence that she began to recognize that she had an impasse early in the problem. Her question about the relationship between two lines of the program is firm evidence that she recognized an impasse.

How Does the Student Search to Localize the Source of Difficulty? First, she tried to hand simulate the program. She failed to do it correctly, narrowing the problem down to the recursive call, and discovered that P did understand the program. Second, she tried to get help from the model, but it did not provide any directly. Third, she tried to build a model, and this led her to focus on the relationship between the recursive call and the inputs.

---

[6](The "mental dialog" given following this footnote is just a plausible guess about S's thinking. It is not something she actually said. However, the data are compatible with the dialog, and very suggestive of most fragments of it.)

What Subgoal Does the Student Set Up to Resolve the Impasse? The subgoal was to understand the relationship between the recursive call and the input line.

How Does the Student Achieve the Subgoal? Her partner showed her how to match the parameters with the inputs. She could use this matching procedure to link the parameters and inputs with her existing knowledge of how to change the values of variables.

What was Learned by the Student? She learned a new element of knowledge that I call "match and change": First, match parameters up with inputs, then change (input variable) to (value of matching parameter) for each input variable and parameter. The new knowledge was constructed in this particular context to solve a particular problem. However, evidence not presented here indicates that she was able to apply this knowledge to other similar problems that involved more than one input.

What is the Student's Final Knowledge State? The final knowledge state is different from the initial state by exactly this new piece of knowledge. The change from initial to final state is shown in Figure 12. Only a few elements of her prior knowledge were touched on by the new knowledge. The jump and execute, optional parameters, and two contexts misconceptions remained in place.

Count-b Impasse, Leading to the Third Knowledge State

The count-b impasse arose because of the limitations of jump and execute when applied to embedded recursion. The next section presents the count-b impasse and the third knowledge state achieved by S. Details are kept to a minimum in order to focus on the changes in knowledge state. Figure 13 shows the program, count-b, which caused the problem. In count-b the recursive call, "count-b (n+1)," is *embedded* in the second line of the program "if." The following dialog gives the flavor of how she made her prediction.

S We have to write a prediction. OK [both looking at the program . . .]. OK well let's say we input 1. Then . . . . . .
S is trying to trace "count-b 1."

P [She looks around at the room, then looks back to the screen.]

S OK well if *n* is less, wait . . . OK well if *n* is less than 5 . . . Then it's gonna do that. [She points to the recursive call, "count-b n+1."]
[There is a long pause.]

S So all it's gonna do is output 5, right? . . . Because before that,
S has apparently run the program all the way through in her head, and has predicted its output.

P If *n* is smaller than 5
P is still working on the individual lines of the program.

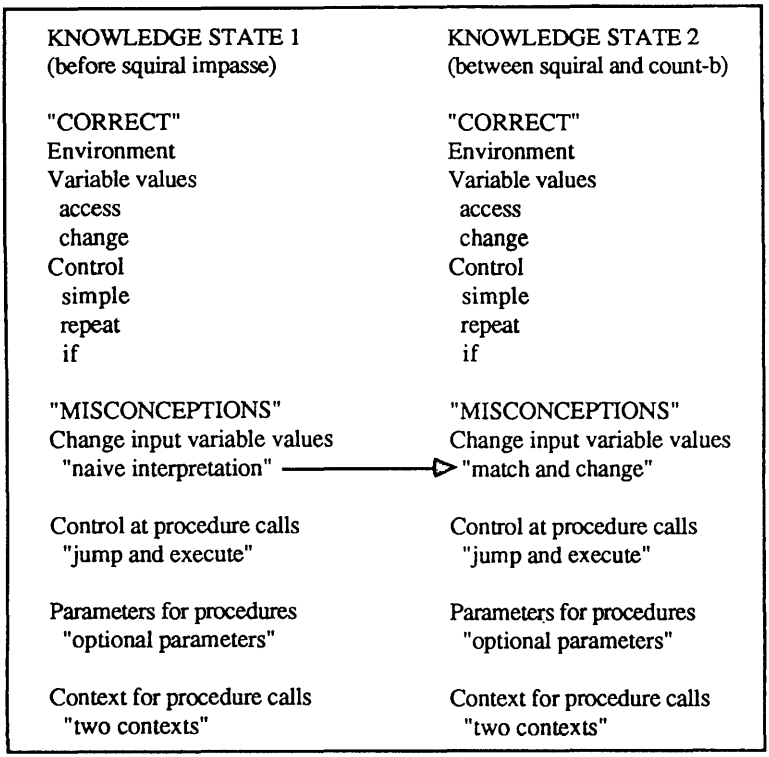


Figure 12. Relevant portions of first two knowledge states.

S Because it says if *n* is smaller than 5,
P Change viewer to *n*.
S Then you'd go back, and start over with a bigger number. Nothing is there that says to output anything. So shouldn't it just, output 5?

S's reasoning is quite plausible. It likely included a little bit more detail; it probably went something like this.

> When count-b 1 is executed, it jumps to the start of count-b, and gives *n* the value 1. If 1 is less than 5 (yes, it is), then do count-b with $n+1$. So jump back to the start of count-b, with *n* equal to 2. This time, if 2 is less than 5 (it is), then do count-b with $n+1$. So jump back to the start of count-b, with *n* equal to 3 . . . This continues until *n* is equal to 5. When *n* is equal to 5, it does not execute the recursive call, and instead goes on to change viewer to 5. So the viewer will be left with the value 5, and the program is done.

Now S ran the program and saw the output, 5, 4, 3, 2, 1. She recognized that she had an impasse. She had mentally run the program, but she had predicted the

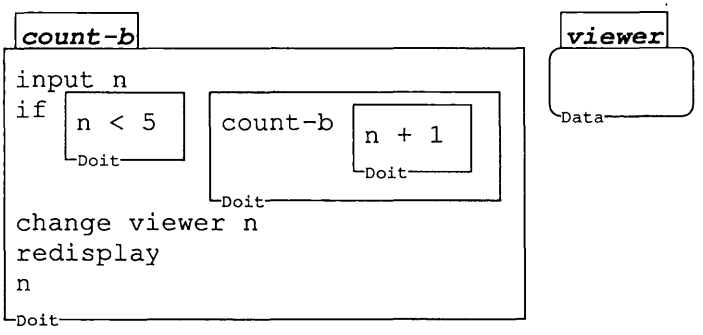


Figure 13. Count-b program.

wrong output. After this, S engaged in considerable search to localize the problem with her knowledge. She ran the program again in her head. She clarified her understanding of the "if" command by checking with the experimenter. Her understanding of "if" was not the source of the difficulty. She traced the program again, checking it carefully against a model. Figure 14 shows the model of count-b that she saw. When I placed the model in the curriculum, I thought that students would immediately see the implications of this model in order to understand how count-b counts backwards. However, S did not see this when she looked at the model. She traced the program to the last level of the model, and still could not see what caused the program to count backwards. The dialog below shows how she set up specific subgoals, again in the form of questions (shown in boldface), and how the experimenter helped her to achieve her subgoals and resolve the impasse.

S OK. I still don't understand what makes it go back to the next level though. Because, let "if n . . ."
S is asking why it goes back to the level where n=4, from the level where n=5.
P OK . . .
S So finally we've changed n to 5, so *n* isn't less than 5. So it doesn't do that [she points to the recursive call].
She correctly points out that it won't make the recursive call at Level 5.
E Uh huh.
S So it changes viewer to 5 . . .
E Uh huh.
S For the first time, and then redisplay
She correctly points out that it will finish the rest of the procedure at Level 5.
E Yep. That just makes it show what's in the viewer, the new value in the viewer.
S Right. So then it shows 5, and then it prints 5 there [she points to the output]. **What is there in there that says it should then go back and change *n* to 4 again?**
She has focused on both of her misconceptions. She thinks that something must tell it

to go back to Level 4 (in reality, the rest of Level 4 is there, waiting to be finished when Boxer returns from Level 5). She also thinks it must change the value of n from 5 back to 4 (in reality, n still has the value 4 at Level 4). Even though the copies of the procedures and their variable n *are shown on the screen, S does not perceive them as copies. At the time, I thought that the implications of all the copies shown on the screen were clear, and so I spent a considerable amount of time trying to focus in precisely on S's difficulty. (See the following dialog.) If I had been part of the dialog all along, as P was on the squiral impasse, I might have focused in on S's difficulty much more quickly.*

(Some repetitive dialog was omitted.)

S **So it should stop, right?**

P Yeah. 'Cause there's no, um, whatever you call it, to tell it to, do it again, do something again.

E OK well it doesn't have to do anything again, but it still has to do what it started. So it's completed this level, right? [gestures over Level 5].

P Uh huh.

E So that level's gone, [he closes Level 5]. Now after it has done this [he points to the closed Level 5], if this is true [he points to "n<5"], then do this [he points to closed Level 5] so it did the recursion.

S But then

E Now it still has this left to do [he points to the rest of the procedure].

He is pointing to "change viewer n" "redisplay" and "n," the last three lines at level 4, where n = 4.

S Wait. **OK, it threw away that level but *n* is still equal to 5 right?**

S seems to have accepted that it still has something left to do. But to her it doesn't make sense that what it has left to do could produce the output of count-b, because n has been changed to 5, and there is nothing to make it change back. This shows how tightly interconnected jump and execute and match and change are at this level. If one holds, it supports the other.

E Not at this level. At this level *n* is still equal to 4 (points to data box *n* containing 4).

Directly confronting the change part of match and change.

S Oh OK. I see.

S begins to work it out for herself.

E It still has to clean up after itself. In other words, it started this level, it still has to complete this level. [He points to rest of procedure at level 4 after "if" line.]

S Right. But now *n* is equal to fou Oh I see so then it just goes and it counts back. [Runs cursor over the "rests" of the rest of the levels.]

S began to confront the same problem at level 4 that she had at Level 5. If "n is now equal to four" then how will it count back 3,2,1? She answers the question for herself by using the same visual argument given for the transition from Level 5 to 4—n does not have to change back, its value has been "preserved" at the level above. Her gesture with the cursor runs across the three last lines of the procedure at levels 4,3,2 and 1.

E Right. Then it does this [he points to the "rest" of level 4]. Then it throws that level away . . .

S Oh I see! I see.

P OK OK
S Do you get it?
P I get it. (She closes Level 4.)
E So now it still has to finish this (he points to "rest" of Level 3).
S I see.

Finally, she focused on two specific features of embedded recursion with the help of the experimenter. When an embedded recursive call is made and returns, the program *must finish* what is left in the procedure. While the recursive call is executing, the original values of variables are *preserved;* therefore, it finishes using the original values.

Roughly, What is the Impasse? S could not understand why she had made an incorrect prediction.

What is the Student's Initial Knowledge State? Figure 12 showed the initial knowledge state for the count-b impasse. It was the same as the final knowledge state for the squiral impasse. The impasse had to do with the jump and execute and match and change concepts. These concepts, which had hitherto worked fine for tracing many programs, were now failing to produce the correct prediction.

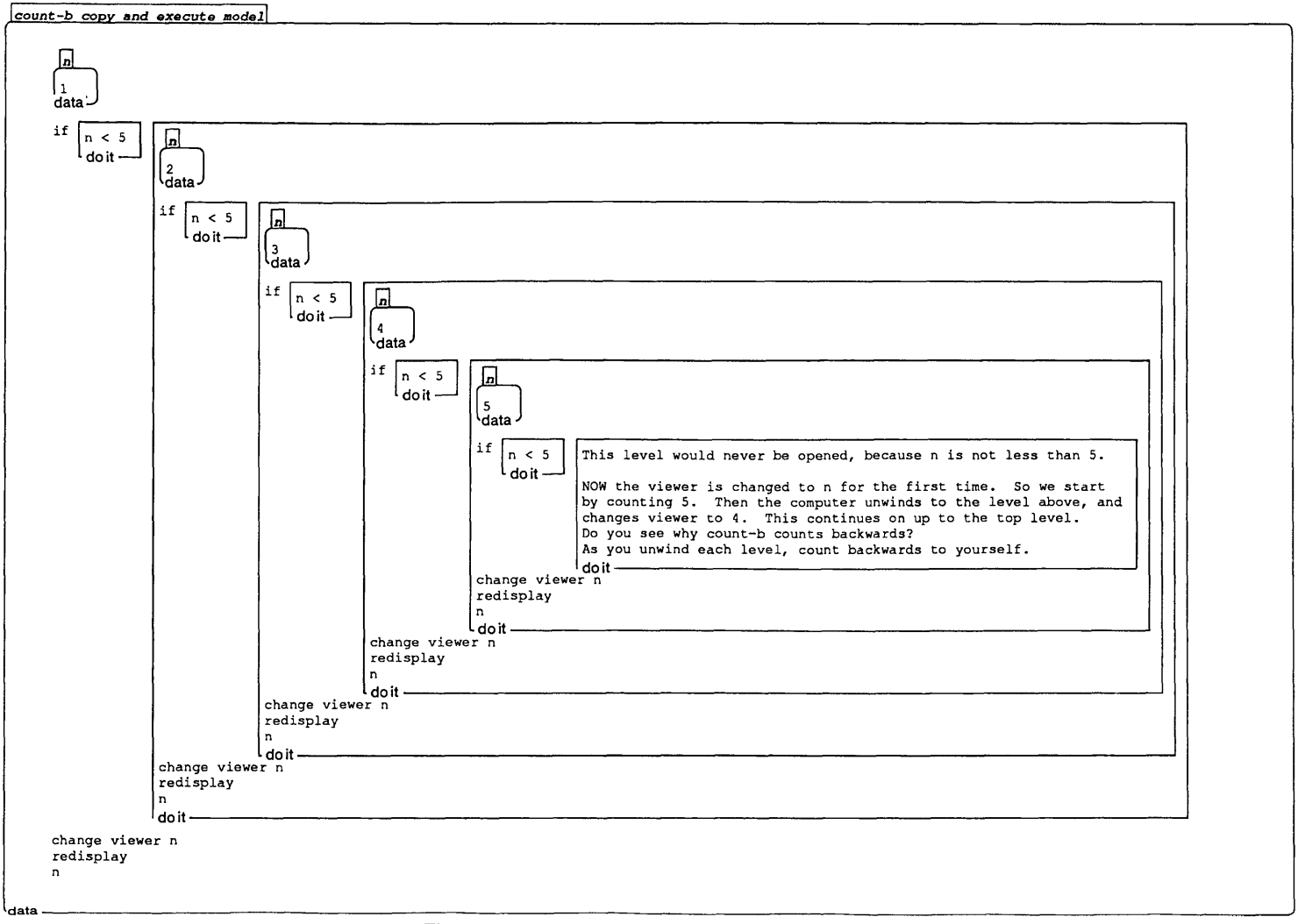
What is the Student's Goal? To make a correct prediction.

Does the Student Recognize an Impasse? Certainly.

How Does the Student Search to Localize the Source of Difficulty? First, she runs the program again in her head. Next, she rechecks her understanding of the "if" command and the "redisplay" command. Finally, she traces the program again very carefully, using the model as a guide. Her search has led her to rule out every answer she can think of. Now the only possibility left is to ask for help.

What Subgoal Does the Student Set Up to Resolve the Impasse? In this case, two subgoals are set up back-to-back. Both are achieved to resolve the impasse. First, she asks, why doesn't it stop, since it has jumped back 5 times (jump and execute) and no longer has any recursion left to do? After the first question is answered, she asks a second: Even if it has to finish what it did at prior levels, doesn't it finish with 5 since *n* has been changed to 5? (match and change). She is saying, in effect, what is wrong with these two pieces of knowledge?

How Does the Student Achieve the Subgoal? The subgoals are achieved with the help of the experimenter, who uses the model to explain. First, jump and execute is shown to need an additional element for the case of embedded recursion. It must return from the recursive call and finish executing anything else left


count-b copy and execute model
n
1
data
if
n < 5
do it
n
2
data
if
n < 5
do it
n
3
data
if
n < 5
do it
n
4
data
if
n < 5
do it
n
5
data
if
n < 5
do it
This level would never be opened, because n is not less than 5.
NOW the viewer is changed to n for the first time. So we start by counting 5. Then the computer unwinds to the level above, and changes viewer to 4. This continues on up to the top level.
Do you see why count-b counts backwards?
As you unwind each level, count backwards to yourself.
do it
change viewer n
redisplay
n
do it
change viewer n
redisplay
n
do it
change viewer n
redisplay
n
do it
change viewer n
redisplay
n
do it
change viewer n
redisplay
n
data

in the procedure. Second, match and change is shown to need an additional element for the case of embedded recursion. The state of input variables is preserved when they are changed in the recursive call, so that when it returns and finishes, it does so with the original values of the input variables at that level. Both of these "explanations" are based on the visual model of recursion in Boxer.

What was Learned by the Student? Two new elements of knowledge appear to have been learned. Both appear to be based on earlier knowledge elements, and both are especially adapted to handle the case of embedded recursion (as opposed to tail recursion, which was handled quite well by earlier knowledge states). I have labelled the first new knowledge element "finish." Finish is an enhancement of jump and execute. Finish says that in the case of embedded recursion, jump and execute may be used to handle the recursive calls, but after the recursive call has been executed, control must return to where the recursive call was made, and anything left in the procedure must be finished. I have labelled the second new knowledge element "preserve state." Preserve state is an enhancement of match and change. Preserve state says that, in cases where finish is going to be used, the original values of variables are preserved while the recursive call is executing. Therefore, when it finishes, it uses the original values. Both finish and preserve state are rather complex knowledge elements that appear to include elements of the Boxer model of embedded recursion, and several other knowledge elements. Once again, S constructed new knowledge which is *sufficient* to resolve the impasse. It is a minimal change to her preexisting knowledge state. The new knowledge is powerful enough to handle many cases of embedded recursion. In fact, it is equivalent to the intended copy-and-execute model except in cases where local variables or inputs are modified within the procedure. S's new structural model of recursion is silent on these cases. Later on (on the rects program),[7] this caused new difficulties for her.

What is the Student's Final Knowledge State? The transition to the final knowledge state is shown in Figure 15. The jump and execute and match and change knowledge elements have been affected by resolution of the impasse. Other aspects of S's knowledge appear to have remained the same. This final knowledge state appeared to be stable throughout the remaining 3 hours of experimentation.

6. DISCUSSION

In this section I discuss the generality of the results, the role of the curriculum, and implications for modelling students' knowledge of program structure, evaluating Boxer, and teaching programming.

[7]Rects is a programming challenge, to create an embedded recursive program that makes a design made up of rectangles.

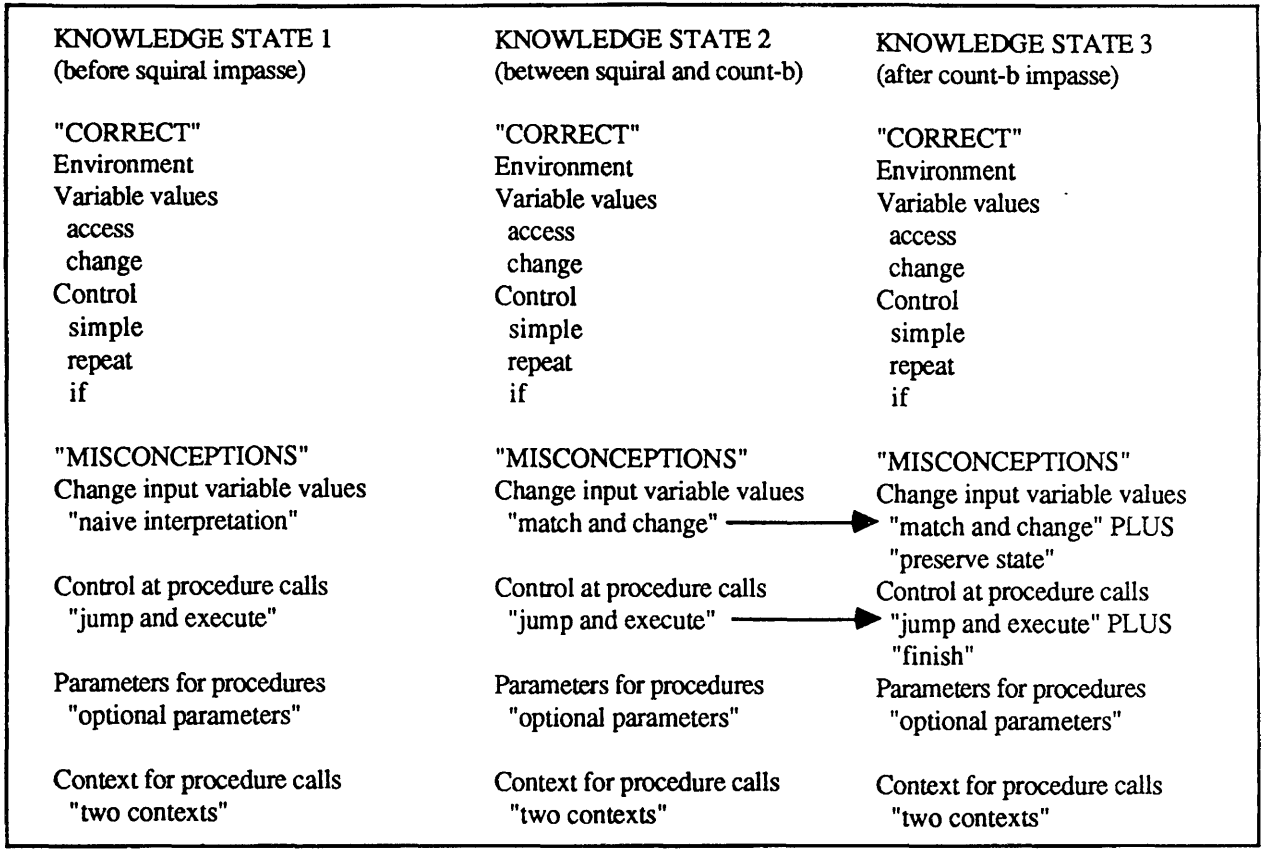


Figure 15. Relevant portions of all three knowledge states.

Generality

How general are the results of this study? In a number of ways, this study has a narrow focus: It concerns the work of one bright, well-motivated student; it is centered around the use of a particular curriculum; and it involves Boxer in a very limited way. On the other hand, this study's results are potentially quite general: Impasse resolution has been claimed to be a broadly encompassing way of viewing learning (Newell, 1990; VanLehn, 1988); the copy-and-execute model of flow of control is central to understanding recursive programs; and, recursive programming is a powerful tool with wide applicability. Further work will be needed to address the question of generality. However, some indications can be drawn from the results obtained here. Each aspect of the following discussion concerns generality in some way.

Role of the Curriculum

There are a number of possible views on my curriculum's influence on the results of this study. Some of the main alternatives include: (1) It failed, except to drop students into impasses that were resolved socially; (2) Its presentation of the copy-and-execute model was inadequate because it didn't get students really to *believe* the model in a general way, but rather to see it as an illustration of the operation of particular programs; or (3) It worked reasonably well. Learning is always piecemeal and in a social context.

I believe that a combination of these views is correct. My instructional goal was not to have students sail smoothly through the curriculum, acquiring the knowledge they need without difficulty. Rather, I hoped that the curriculum would serve as a kind of "scaffold," structuring students' exploration of the domain, giving them plenty of opportunity for success, and providing some help, particularly in the form of the copy-and-execute models, for understanding the structure of recursive programs. I did not want to avoid social learning factors, and, in fact, hoped that the students would cooperate, and expected them to ask me for help occasionally. I was quite pleased with the engagement of the students in the problems, both tracing example programs and programming the challenges. They clearly seemed to value structural knowledge of Boxer.

To improve the curriculum, my main goal is to make the presentation of the copy-and-execute model more effective. I plan to do this in two ways: first, to present it in situations where it clearly provides a correct explanation of a program's behavior (not with tail-recursive programs as in this curriculum); and second, to build *dynamic* models of the execution of procedures, rather than using the static models provided in this curriculum. I hope that these efforts will help students to acquire the copy-and-execute model more efficiently and to apply it more broadly.

Modelling Students' Knowledge of Program Structure

Although S's knowledge has become considerably more powerful as a result of resolving two impasses, it is still far from the knowledge we might expect an

expert to have. S's knowledge corresponds to what diSessa (in press) calls a "distributed model"; diSessa describes distributed models as follows:

> These are something like a learner's spontaneous versions of structural models. That is, learners try to provide general descriptions of language constructs that explain why and how they do what they do. Naturally, these descriptions will typically fall far short of the intended structural models, with much more diversity in their sources and unreliability in their transfer to other situations. Indeed, I call these *distributed models* to indicate the diverse and *ad hoc* "principles" that serve as the basis for them.

Impasse resolution appears to be a good framework for describing how S constructed her distributed model of recursion. Each impasse involved a particular problem-solving context. In order to resolve the impasse, she constructed a particular piece of knowledge. Each new piece of knowledge addressed a separate aspect of her overall understanding of recursive programs. She *accumulated* a better overall understanding through resolution of each local problem.

There are many open questions concerning students' knowledge of program structure. Do students always develop such a complex distributed model, or are there other ways of learning structural knowledge that result in cleaner and more coherent knowledge structures? In particular, might it be possible through redesign of the curriculum to get students to acquire the copy-and-execute model more quickly, and to apply it more effectively? Do experts have clean and coherent structural models, different in kind from those of students? Or do experts simply have more elaborated distributed models that cover more territory and contain greater detail? How does knowledge of program structure relate to other types of knowledge about programming, such as functional knowledge, plans, templates, and strategies? I will pursue these questions in further research.

Evaluating Boxer

We would like to determine whether Boxer's design (especially its visualizable and complete structural model) does achieve its stated goals of making programming more comprehensible and learnable (diSessa & Abelson, 1986). Although this study is not an evaluation study, it does suggest that (1) many aspects of Boxer had been successfully learned by the student before the study, such as navigation of the environment, several forms of flow of control, and access and modification of variable values; and (2) Boxer is representationally adequate to display its own structure in models for discussion among students and instructors. Much work remains to be done in evaluating Boxer.

In order to evaluate Boxer's comprehensibility and compare it with other languages, a *cognitive benchmark* (diSessa, in press) is in the process of being constructed. A cognitive benchmark is the result of testing students' understanding or learning of programming in a principled way. The term *cognitive* is used

because the results concerning learnability and comprehensibility are of specific interest, rather than more traditional benchmark results concerning speed, size of data structures, and so on. The term *benchmark,* rather than *test* is used, because it invokes a main feature of benchmarks: They are repeatable across a wide range of systems. Because they can be repeated at will, they can serve as a basis for comparison. Benchmarks also tend to take on the status of a goal to be reached or surpassed. If well designed, they can, therefore, spur healthy competition among systems.

The most obvious way of comparing languages is to teach the same programming topics in both languages, and then give the students a test that measure how well they know some aspect of programming. If one group does better than the other, then *all other factors being equal,* it is likely that the better structure is that of the language in which the students learned more. The difficulty with this proposition lies in the phrase, "all other factors being equal." What are the relevant factors which must be controlled? Certainly the students' backgrounds, the topics taught, and the instructional methods should be. Schweiker and Muthig (1986) carried out such a study. They controlled for these factors and gave students similar problems involving data construction (creating complex data structures from simpler ones) in Logo and Boxer. They found that students were three times faster in solving such problems in Boxer than in Logo. These results are encouraging to Boxer proponents because they attribute the faster performance of the Boxer students to its concrete visible structure. However, Boxer proponents are not completely sanguine because, in the Schweiker and Muthig study, data construction was taught in isolation. There is no guarantee that these results will not be "washed out" through interaction with many other effects when Boxer and Logo are used in a broader context.

This study points to a serious potential problem with the approach previously outlined. The student in this study developed a distributed model of the structure of recursive programs that was quite particular to her problem-solving and learning trajectory. Although the point was not emphasized, it should have been clear from the dialogs presented that the two students' understandings differed substantially from one another. Each student played a large role in defining her own learning experience, and her active participation interacted in complex ways with the curriculum, the structure of the language, and the social context. There is no way to "control" for all of these factors and the interactions among them.

The research group in which I participate is taking a different approach to developing a cognitive benchmark. It is developing a framework for the cognitive benchmark in order to support it theoretically, empirically, and pedagogically. Theoretical support includes a comprehensive view of other types of programming knowledge, modes of using that knowledge (such as analysis, synthesis, and debugging), and models of learning processes (such as impasse resolution). It hopes to use the theoretical framework to understand how the learning of structural knowledge fits into learning programming as a whole, and

thereby to sort out the contributions of many factors either to furthering or hindering the learning of structural knowledge. Empirical support includes rich models of the use of this benchmark and interpretation of results. Pedagogical support includes comprehensive teaching materials for the structure of Boxer. These materials reflect the belief that testing and teaching can go hand in hand. Through use of this benchmark, it hopes to refine the framework, and to develop detailed models of learning trajectories for Boxer. The goal is to make the framework general enough to be able to apply it to any procedural programming language, hopefully to establish a basis for construction of a benchmark that can be applied to a variety of languages.

Educational Implications

Several related educational implications are implicit in my perspective and demonstrated in this study. First, I believe that students bring extraordinary competence with them into any learning situation. The ability of these 2 eighth-grade girls to navigate independently a complex environment, form their own goals and strategies, and learn a difficult topic in programming, is remarkable. It would be easy to focus on these students' "misconceptions"—especially on their incorrect or limited knowledge of recursive program structures—some of which persisted at the end of the study. Instead, I focus on their successes. They took control of their own learning and are in a good position to learn more. (A follow-up study indicates that they are indeed learning more.) I believe that all students have competences, which can and must be capitalized on, in order for them to learn.

Second, these students entered with misconceptions, constructed misconceptions, and left with misconceptions. This might be viewed as a failure. Instead, I look upon it as a natural part of the learning process in complex domains. A student is not a *tabula rasa,* an empty vessel into which correct knowledge may be poured. Instead, students construct their own understanding to serve their own goals. Their constructions, based on limited experience, will necessarily be limited or incorrect in some way. However, in serving their own goals by constructing intermediate (rather than wrong) conceptions, the students experience successes and become prepared to adopt larger goals, and thereby, develop improved understanding.

Boxer is designed to make programming so accessible and useful that it can be used for learning and exploration in a wide array of mathematical and scientific domains. This doesn't mean that learning Boxer is expected to be *easy*. Like any worthwhile and complex problem-solving skill, it is expected that learning to program in Boxer will require sustained efforts over a long period of time by teachers and students. This article suggests that such sustained efforts can pay off in a small but significant way over a short period of time. Some of the long-term patterns of learning and using Boxer are currently being designed and charted. As part of this effort, I am rewriting the curriculum that was used in this study, and integrating it with a larger effort to teach Boxer in a systematic way.

7. CONCLUSIONS

In this study I examined the learning of 1 eighth-grade girl as she learned more about the structure of recursive programs in Boxer. I used impasse resolution both as an analytic frame for understanding learning, as well as the basis for an empirical method, impasse analysis. Impasse analysis was used to chart the construction of new knowledge by the student in fine detail. The student made progress in overcoming two misconceptions about recursive programming. These two misconceptions were identified, in general, by Kurland and Pea (1989), as "decontextualized interpretation of commands," and "mental model of embedded recursion as looping." The student's progress occurred when impasses arose and were resolved in her problem solving. Knowledge learned by resolving impasses was specific to the conditions that caused the impasse and the way in which it was overcome. In similar later conditions, the student avoided impasses using the knowledge she had learned. At the end of the study the student had a better understanding, but it was still not entirely "correct" in ideal terms. Some aspects of her faulty understanding were not addressed at all. I discussed the implications of these results for modelling the learning of this type of knowledge, evaluating Boxer, and teaching computer programming. Next steps in this work include: (1) an attempt to use impasse analysis to chart the learning of other kinds of programming knowledge; (2) development of a cognitive benchmark test to assess students' knowledge of Boxer's structure; and (3) construction of an improved curriculum for teaching Boxer programming.

REFERENCES

diSessa, Andrea (in press). Local sciences: Viewing the design of human–computer systems as cognitive science. In J.M. Carroll (Ed.), *Designing interaction: Psychology at the human–computer interface*. New York: Cambridge University Press.

diSessa, Andrea, & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM, 29,* 859–868.

Kurland, D.M., & Pea, R.D. (1989). Children's mental models of recursive Logo programs. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, NJ: Erlbaum.

Laird, J., Rosenbloom, P., & Newell, A., (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning, 1.*

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Schweiker, H., & Muthig, K. (1986). Solving interpolation problems with Logo and Boxer. In P. Garny & M. Tauber (Eds.), *Visual Aids in Programming*. Heidelberg: Springer.

Van Lehn, K. (1988). Toward a theory of impasse-driven learning. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligence tutoring systems*. New York: Springer Verlag.