INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

An investigation of a manipulative simulation in the learning of recursive programming

by

Randall Wayne Bower

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Co-majors: Computer Science; Education (Curriculum and Instructional Technology)

Major Professors: Rex Thomas and Ann Thompson

Iowa State University

Ames, Iowa

1998

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Randall Wayne Bower

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Co-Major Professor

Signature was redacted for privacy.

Co-Major Professor

Signature was redacted for privacy.

For the Co-major Program

Signature was redacted for privacy.

For the Co-major Program

Signature was redacted for privacy.

For the Graduate College

TABLE OF CONTENTS

INTRODUCTION

As computers continue to permeate society, computer related skills are more valuable and necessary. For many, the required skills have moved beyond use of a single, specific program to a more complete sense of computer literacy. Computer users need to be proficient with a variety of applications such as word processing, spreadsheets, database packages, desktop publishing, and internet tools. However, becoming proficient with more than one or two of these applications is a challenging task due in large part to the constant changes and updates made in each package. Thus, it is important that users develop the ability to use the computer as a learning tool and be able to adapt to changing and new software packages as the need arises.

There is also a need for more users to move beyond creating documents, managing data, and generating reports. Computer users must develop an understanding of how to use the computer as a problem-solving tool. To this end, there is increased attention being paid to teaching programming. Seymour Papert (1980), among others, claimed that there is significant transfer from programming skills to general problem solving skills. He also argued that programming itself is a way of thinking and thus a tool for solving problems. The difficulty is that programming is a complex task. It typically takes years of schooling and years of experience to become an expert programmer. It is not realistic for a non-computer science student studying programming as a means of developing general problem solving skills to take more than one or perhaps two introductory programming courses. Thus, important questions are what topics to teach in these introductory courses and how to teach those topics. It is important that the answer be valid for both computer science and non-computer science students, as it is not practical in many institutions to design two separate courses for each group of students.

One topic that is often on the edge of being included in introductory programming courses is recursion. Daniel McCracken (1987) answered the question, "Is recursion an advanced topic?" as follows (p. 3):

> Absolutely not. Recursion is fundamental in computer science, whether understood as a mathematical concept, a programming technique, a way of expressing an algorithm, or a problem-solving approach...Recursion is a powerful tool for teaching computer science - it can make later topics more understandable. *Of course* you can show linked list algorithms using iterative techniques, but the recursive methods are *so* much simpler!

These statements indicate that it is important to present recursion early, both for non-experts and experts-to-be. However, even though recursive computer programs can solve complex problems with elegant and simple solutions, teaching students how to program recursively has proven to be a challenging and elusive task. The struggle begins with defining recursion. Programming constructs such as sequencing, conditionals and iteration have somewhat naturally occurring examples in our everyday lives. For example, a cake recipe might include statements such as, "First mix the dry ingredients, then add eggs and milk. Stir until well mixed. If batter is thick, add more milk." Such examples can lead to direct and helpful analogies. Recursion on the other hand does not lend itself to such commonplace and helpful examples.

As noted, recursion is at once a mathematical concept, a programming construct, and a problem-solving method. The main criteria for something to be recursive is that its definition must refer to itself. The

Fibonacci sequence is commonly used to introduce recursion as it is easily written in mathematical notation and as a computer program. The Fibonacci sequence is an infinite sequence of integer values. The first few values are 1, 1, 2, 3, 5, 8, 13, etc. The sequence is defined by the following rules: (1) the first two values in the sequence are 1; (2) all other values in the sequence are calculated as the sum of the previous two values. In mathematical notation, this is written as:

$$f_n = \begin{cases} 1 & \mid n = 1 \text{ or } n = 2 \\ f_{n-1} + f_{n-2} & \mid n > 2 \end{cases}$$

A function to calculate the n^{th} Fibonacci number could be written in a programming language as:

```
int fibonacci( int n )
{ if( n <= 2 )
  { return 1;
  }
  else
  { return fibonacci( n-1 ) + fibonacci( n-2 );
  }
}
```

Notice that in both definitions, if n is greater than 2, the definition refers to itself. Determining the value of f_{n-1} or `fibonacci( n-1 )` requires looking again at the original definition, this time to find the correct value for `n-1`. This is called the recursive case. Also notice that there is another case in each definition, namely when n is equal to 1 or 2. In this case, the answer is known and there is no need to refer to the original definition. This prevents the self-referencing process from continuing indefinitely. This is called the base case. The last thing to notice is that since the recursive case refers to the original definition with a smaller value for n, the base case will eventually be reached and the process will stop. This is the heart of recursion: determining how the nature of a problem lends itself to being divided into two or more cases with at least one base case and each recursive case moving in some manner closer to a base case.

Although the Fibonacci sequence is perhaps easy to understand once presented, something more is needed to help students visualize the recursive nature of a problem and then create their own solutions. While there are some naturally existing examples of recursion in our lives, such as the image produced by pointing a video camera at the monitor it is attached to, they only capture the idea of self-reference. What is needed is something to help students identify the recursive structure of certain problems and then identify the critical features of recursive solutions to those problems. Computer simulations have shown much promise in providing students with rich experiences that bring them into contact with the critical features of a topic (Hooper and Thomas, 1990, Upah and Thomas, 1993, Brandt, Hooper, and Sugrue, 1991). When these experiences are presented before formal classroom instruction, they provide groundwork for the formal discussion and make it more meaningful. The current research project develops and evaluates a simulation that provides students experience with recursive problems which will make future classroom discussion of recursion more meaningful.

Statement of the Problem

Recursion is a fundamentally important topic in computer science. Even so, it is often omitted in introductory courses, or discussed only briefly. This is likely due, at least in part, to the fact that teaching recursion has been difficult. However, successful simulations have shown that the computer may hold the key to solving this problem. The challenge is to develop an appropriate simulation and lesson plan for introducing recursion to students early in their programming experience.

Purpose of the Study

The purpose of this research was to develop a computer simulation to aid in learning recursive programming. The study was motivated by the extreme difficulty many novice programmers have with this topic and the successful employment of simulations in the learning of other computer programming topics. The goal of the simulation was to help students gain insight into recursion before the topic was formally examined in class. The data gathered during this research helped bring into sharper focus the specific areas of recursion that are most problematic and provides evidence for determining the strengths and shortcomings of the simulation in addressing these areas.

Research Hypotheses

Data were collected from two groups of students enrolled in an introductory, college-level programming course. One group used the simulation as an introduction to recursion, and the other group received a more traditional, lecture-based introduction to recursion. Students in both groups then completed a set of eight recursive programming exercises. The first four programming exercises were recall exercises since they were the same as the first four exercises in the simulation and were used as examples in the control group lecture. The next three exercises were application exercises since they were different from the first four, but required the same type of solution. The last exercise was a transfer exercise since it required students to extend what they had learned to a new situation. The set of exercises and the lesson plan are described in detail in Chapter 4.

Students in both groups were also given several recursive problems on the final exam. The recursive problems on the final exam contained some recall, some application, and a transfer problem. The recall and application problems were similar to those in the simulation. The transfer problem was the traditional Towers of Hanoi problem. These problems and grading criteria can be found in appendices A and B.

Finally, those students enrolled in the second programming course the following semester were given a follow-up exam. This exam consisted of four problems. Two problems were similar to those in the simulation, one problem asked students to trace a recursive function, and one problem required writing a recursive function to calculate the n^{th} Fibonacci number. This exam and grading criteria can be found in Appendix C.

The following specific hypotheses were tested:

1. Students in the experimental group will successfully complete more exercises.
2. Students in the experimental group will successfully complete all exercises in fewer attempts.
   A. Students in the experimental group will successfully complete the recall exercises in fewer attempts.
   B. Students in the experimental group will successfully complete the application exercises in fewer attempts.
   C. Students in the experimental group will successfully complete the transfer exercise in fewer attempts.
3. Students in the experimental group will score higher on recursive programming questions on the final exam.
4. Students in the experimental group will score higher on recursive programming problems on a follow-up exam given the following semester.
5. Students in the experimental group will use different strategies when solving recursive programming problems.
6. Learning style will not affect the number of completed exercises.
7. Learning style will not affect the number of attempts required to complete the exercises.
8. Learning style will not affect student performance on exam questions.

Limitations of the Study

This study was conducted in view of the following limitations:

- The subjects of the study were drawn from two sections of one course during one semester, which limited the sample size.
- The subjects were placed into the control and experimental groups based on their course section rather than being randomly assigned to each group.
- Due to the open lab setting, some interaction between students was unavoidable.
- The simulation software was untested on beginning, college-level programmers.
- Only a limited amount of time in the course could be devoted to recursion and this research. This limited the number of recursive programming tasks that could be assigned to the subjects.
- The primary researcher did all grading.
- The primary researcher conducted all class sessions with both the experimental and control groups. The researcher's work in developing the simulation and discussion of recursion with the experimental group influenced, and likely improved, the lecture and discussion in the control group.

LITERATURE REVIEW

In this chapter, literature on teaching and learning recursion and research on the use of computer based simulations is reviewed. The literature on teaching and learning recursion can be divided into three broad categories: lesson plans, mental models, and programming environments. Traditional lesson plans include detailed methods for presenting recursion. These usually begin with numerous non-computer examples of recursion. Mental models are an external representation of student's struggle to come to understand recursion. Recursive programming environments are in the form of tutoring systems, and often attempt to help students form appropriate mental models. After discussing each of these areas, a brief review of the research on simulations used to introduce other computing concepts is presented. Based on this literature, critical features of recursion are identified in order to guide the design and discussion of the recursion simulation.

Defining Recursion

A significant reason for the difficulty in teaching and learning recursion is that recursion may not be well-defined (Give'on, 1990). The most simplistic definitions say only that a process or object is recursive if it refers to itself. More complex definitions of recursion list the critical features as: (1) the existence of base or trivial case(s); (2) the existence of recursive case(s) which break a larger problem into smaller, similar problems; (3) the flow of control as recursive calls are made; and (4) the notion that the recursive case must move in some way closer to the base case. However, such definitions fall short of actually explaining the process of recursion, both the process of how a recursive definition or program works and the process of creating a recursive definition or program. To understand the process of recursion and to be able to write a recursive definition or program one must be able to visualize the nature or structure of a problem and how solutions to smaller, similar problems are combined to solve the original problem.

Adding to the difficulty is the fact that recursion comes in many forms: tail, embedded, procedural, and functional. To compare these types of recursion, consider a Logo procedure to draw a square. In Logo, the programmer is in control of a turtle, which is initially placed in the middle of the screen. The turtle can be told to move forward or backward a certain number of pixels and as is moves it draws a line from its starting point to its ending point. The turtle can also be told to turn a given number of degrees right or left. Thus, to draw a square with each side measuring 100 pixels, the turtle would need to be given the following list of commands:

```
forward 100
right 90
forward 100
right 90
forward 100
right 90
forward 100
right 90
```

The final `right 90` is not strictly necessary, but it has the effect of returning the turtle to its original location and direction, so it is usually included. A procedure to draw such a square could be written as follows:

```
to square :size
  forward :size
  right 90
  forward :size
  right 90
  forward :size
  right 90
  forward :size
  right 90
end
```

With this procedure defined, the programmer can cause the turtle to draw a square with sides of 100 pixels by issuing the command `square 100`. The `100` is taken from the command and placed in the variable `:size`. The turtle then executes all of the commands until the `end` is encountered. When the `end` is encountered, control is returned to whatever issued the command. Sometimes this is the keyboard if the programmer typed the command directly; sometimes this is another procedure that has called the `square` procedure. For example, consider the following procedure to draw a house:

```
to house :size
  square :size
  forward :size
  triangle :size
end
```

Assuming the `triangle` procedure has been properly defined, this procedure will draw a simple house. When the command `house 100` is given, the value `100` is taken from the command and placed in the variable `:size`. Then, since the `square` command is the first command in the `house` procedure, control is passed from the `house` procedure to the `square` procedure, along with the value stored in the variable `:size`. While the `square` procedure draws the square, the `house` procedure is suspended, waiting for the `square` procedure to finish. When the `square` procedure is finished, control is passed back to the `house` procedure, which then issues the `forward` command. When that is finished, the `house` procedure issues the `triangle` command and again suspends and waits for the `triangle` procedure to finish executing. Finally, when the `triangle` procedure is finished executing, control is again passed back to the `house` procedure where the `end` statement signals that the procedure is finished. This flow of control and the idea that one procedure will suspend and wait while another procedure executes is important in understanding recursion.

A tail recursive version of the `square` procedure would be written as follows:

```
to square :size :sides
  if :sides = 0 [ stop ]
  forward :size
  right 90
  square :size :sides-1
end
```

Notice that the last line of the procedure is another call to the `square` procedure. To draw a square of size 100, this procedure would be invoked with the command `square 100 4`. The `100` is again taken from the

command and placed in the variable `:size`. The `4` is also taken from the command and placed in variable `:sides` to indicate that four sides must still be drawn. First, the value in the variable `:sides` is checked to see if it is zero. If it is, the procedure stops. Otherwise, the turtle is told to move forward and turn right, drawing one side of the square. The last command in the procedure then tells the turtle to draw a square of the same size with one fewer sides. That is, the second parameter is calculated as the current value in the variable `:sides` minus one. When this happens, the currently executing `square` procedure suspends and waits for the newly called `square` procedure to finish. That is, there are two separate copies of the `square` procedure. The one that was told to draw a square with four sides is suspended, waiting for the one that was told to draw a square with three sides to finish.

Since the variable `:sides` is initially assigned a value of `4`, it will take four recursive calls to reach the case where `:sides` equals zero and the procedure stops. However, recall that each of the procedures that made a recursive call is still waiting for the procedure it called to finish. Thus, control must be successively passed back to each of these waiting procedures so that they can properly end.

Students have traditionally had less trouble understanding tail recursion because they are able to view it as a form of looping (Kurland and Pea, 1985). This view of recursion ignores the fact that as each recursive call is made the calling function suspends its execution and waits for the called function to finish. The reason students are able to ignore this important detail and still correctly interpret and even write tail recursive code is that once the suspended functions resume execution, by definition of tail recursion, there is no further code to be executed. Thus, to the student, it is not evident that the suspended functions are ever resumed. (In fact, some interpreters treat a tail recursive call as a 'goto' statement.)

In contrast, with embedded recursion there is more code to be executed in the calling function after the recursive call. That is, once the called function finishes executing, the calling function resumes executing with either the next line of code or the surrounding expression. If several recursive calls are made before a base case is reached, there will be a significant amount of execution to be done as the recursion unwinds. For example, consider the `square` procedure with the commands `fd :size` and `left 90` added after the recursive call:

```
to square :sides :size
  if :sides = 0 [ stop ]
  forward :size
  right 90
  square :sides-1 :size
  forward :size
  left 90
end
```

At first glance, one might think this would draw a stair-step pattern since each call to the procedure moves forward, turns right, moves forward again, and then turns left. This is not the case due to the fact that the calling procedures suspend their execution while the recursive call executes. Thus, only the `forward` and `right` turn commands are executed until a base case is reached, drawing a square in the same fashion as the original square procedure. Then, the procedure that reached the base case terminates and the procedure that called it resumes

executing with its `forward` and `left` turn commands. This procedure will then terminate and the procedure that called it will resume executing with its `forward` and `left` turn commands. This continues until the original procedure resumes, executes its `forward` and `left` turn commands, and finally terminates the whole process. The result is another square to the left of the first square. Students have a difficult time understanding this flow of control and often ignore the code that appears after the recursive call (Kessler and Anderson, 1989). This may be because they mistakenly think the `stop` command stops everything rather than just the currently executing procedure.

There is also a significant difference between procedural recursion and functional recursion. Procedural recursion performs a task, such as the Logo procedure to draw a square. Functional recursion computes a value, such as the function to compute a Fibonacci number presented in the previous chapter. Procedural recursion has been shown to be easier for students to understand since tasks like drawing a shape in Logo are somewhat familiar (Troy and Early, 1992). Functional recursion has proven to be more difficult. As with embedded recursion, this seems to be due to a lack of understanding of the flow of control. Students have a difficult time putting the values returned from each recursive call together and computing a final result as the recursion unwinds (Anderson, Pirolli and Farrell, 1988).

In addition to the fact that there are many different forms of recursion as described above, there is also not a clear definition of what it is to "understand" recursion. Some researchers have given students a recursive definition of a mathematical function and tested them to see if they could compute different values for the function (Anzai and Uesato, 1982). Other researchers have tested students to see if they could trace the execution of recursive code and predict the results (Widenbeck, 1988). Few have tested students' ability to actually write recursive code, as illustrated in the following discussion of recursion lesson plans.

Lesson Plans

A series of articles in the December 1983, issue of The Computing Teacher illustrates the traditional lesson plan approach to teaching recursion. The first (Riordon, 1983) and most complete was a three part lesson plan that began with non-computer recursive experiences, then moved to the computer with some simple examples, and finished with a discussion of embedded recursion. In part one, Riordon claimed simply that "if students are to understand the concept of recursion, then that understanding should be grounded in a rich background of non-computer experience" (p. 38). The first non-computer experience with recursion is the popular example of standing between two mirrors, looking at the reflection in one mirror, which is a reflection of the second mirror. In the second mirror is a reflection of the first mirror, and so on. Students are asked to comment on what they see and attempt to verbalize the process that produces the infinity of images. However, Riordon cautions teachers "not to over-emphasize the verbalizing aspect; your goal is to produce a right-brain understanding, not a left-brain verbalization" (p. 39) and that "your purpose here is to *start* students thinking - not to have them *finish* thinking" (p. 40). Other examples from part one include pointing a video camera at the

monitor it is attached to, recursive drawings, and a tape recording that has a copy of itself spliced into the middle, the copy of course having another copy spliced into the middle of it, etc.

These examples are intended to provide students with experiences that they can associate with the word "recursion". In preparation for moving to the computer, part two of the lesson begins by having students play a simple game. Their instructions are whenever you hear the word "pow", raise your arms, lower your arms, and say "pow". Students are asked to play the game until they realize it will go on forever. They are then asked to think about what happens, and as they do so the teacher writes the instructions on the board. When the instruction "Say 'pow'" is written on the board, the teacher draws an arrow and begins writing another complete copy of the instructions. Riordon notes carefully here that "it is important to use the word 'copy' and to actually write in the copy. What you should not say is that it 'goes back up' to the beginning" (p. 60). These types of examples continue at the computer with Logo programs to draw a box and a polygon. A slight twist is added by passing the recursive procedure a parameter that increases with each call, thus drawing an ever-increasing sized shape. Finally, part two concludes by considering the problem that these procedures never stop, and introducing conditionals for "taming" recursion. In all of these examples, students are presented with the recursive code and are asked to determine what happens. They are encouraged to write out each copy of the code with the appropriate values for the parameters, and arrows connecting one copy of the code to the next.

Part three of the lesson moves to embedded recursion. The activities are similar to those in part two, tracing Logo procedures that draw shapes, making copies of the code with the parameter values filled in and arrows connecting the copies. Towards the end of the lesson, two numerical examples are presented to compute powers and factorials. At no point in the lesson are students given the task to write their own recursive procedures to draw shapes or functions to compute values.

Other articles in the series follow this same basic pattern: non-computer examples of recursion, some simple recursive Logo procedures, and detailed discussion of how the computer executes these examples (Billstein and Moore, 1983; Lough, 1983; Moore, 1983). Each author spends the majority of their effort discussing the flow of control as recursive procedures execute. There is little mention of testing student understanding of recursion, and exercises that are described only have students trace a recursive procedure and predict the results.

Almost ten years later, The Computing Teacher published another two-part lesson plan for teaching recursion (Troy and Early, 1992). The basic structure of the lesson is the same as those previously described. The first part provides students with numerous non-computer examples of recursion, most of them the same. The lesson then moves to a discussion of recursive pseudocode, which again centers on writing out copies of the code and drawing arrows to show the flow of control. Finally, the lesson discusses recursive procedures and then functions, but students are again only presented with examples and asked to trace their execution and predict their results. Little had changed from the lesson plans described in 1983.

Troy and Early (1992) do note that "the abstraction inherent in recursion can make the process difficult for both students and teachers" (p. 25). However, the traditional lesson plan they describe focuses on explaining

in great detail how recursive procedures execute and does little to teach the student how to make the necessary abstractions to deal with recursion. Ford (1984) describes what he calls an "implementation-independent" approach to teaching recursion in which the goal is to use abstraction to simplify recursion. The purpose of his article was to "argue that the demonstration of the correctness of recursive solutions also belongs at the very highest levels. Specifically, it will be shown that the correctness of a recursive algorithm can be shown without appealing to implementation details" (p. 213). The paper then describes and demonstrates a method of formally proving the correctness of a recursive algorithm that is similar to mathematical proof by induction. There is nothing included in the paper on how this knowledge will help novice programmers learn to write recursive programs.

An interesting point of debate among researchers has been the influence of learning iteration on future learning of recursion. Anzai and Uesato (1982) performed a study in which 88 middle school children studied the definition of the factorial function. There were two versions of the factorial function, one iterative and one recursive. The students were divided into two groups with both groups being presented both versions, but the order of presentation was reversed. Their conclusion was that "recursive procedures may be acquired based on learning of the corresponding iterative procedures" (p. 100). However, they do note that this conclusion was based only on looking at the mathematical definition of the factorial function, and that "we should be cautious when we try to extend the consideration to more complex domains such as computer programming" (p. 102).

Wiedenbeck (1988) was critical of Anzai and Uesato's results because they only used two groups, iterative-recursive and recursive-iterative. Wiedenbeck repeated Anzai and Uesato's study, again using the mathematical definition of the factorial function, but this time using two additional control groups, iterative-iterative and recursive-recursive. The results did not support Anzai and Uesato's because students in all groups performed better on the second set of exercises regardless of which method they had used first. Wiedenbeck preformed a second study similar to her first, but this time working with iterative and recursive computer programs to compute factorials instead of just mathematical definitions. In this experiment, her results suggest that prior experience with the iterative version may facilitate understanding of the recursive version, to some extent. However, it should be noted that "understanding" here is simply the ability to correctly predict the result produced by the code. Thus, previous understanding of the factorial function itself may have been what facilitated future understanding of the recursive version.

Kessler and Anderson (1986) looked at the transfer of skill between writing iterative and recursive procedures. Their results demonstrated that previous experience with either iterative or recursive programs facilitated later writing of similar recursive programs, but previous experience with recursive programs actually slowed down later writing of similar iterative programs. However, a closer look at the procedure for these experiments is warranted. The authors give a detailed description of the instructions given to students: "The instructions about how to write the recursive or iterative functions emphasized three aspects. First, a theoretical explanation of the recursive or iterative construct was given. Then, a template for writing the function was presented. Finally, the subjects obtained an example of a recursive or iterative function" (p. 140). With all of

this provided, especially the template for each problem, they noticed a significant amount of random behavior, or guessing. Thus, gains made by students who studied iterative solutions first may have simply been the result of more experienced guessing.

The result that experience with recursion actually slowed down later learning of iteration is quite curious. Kessler and Anderson (1986) claim that this is because students who studied recursion first developed such a poor mental model of it that it actually hindered their study of iteration. Students are perhaps able to make better guesses with iteration than recursion. With iteration, even poor guesses will usually result in the loop executing a few times and students can usually tell if it executed too many or two few times. If their next guess causes the loop to iterate more or fewer times, and that is closer or farther from the correct number of iterations, they can tell if they are getting closer or farther from the correct solution. Students may or may not become proficient with iteration in this manner, but they are at least able to see some partial results and gain a basic understanding of how a loop executes. Recursion on the other hand does not lend itself to such incremental development. If students do not understand the flow of control when a recursive call is made, for example, it is quite difficult for them to realize that their guess caused the solution to make too many or too few recursive calls. Thus, their next guess is no more informed than the previous one.

Mental Models

There are numerous mental models of recursion described by various researchers (Anderson et al., 1988; Bhuyian et al., 1989, 1991, 1992, 1994; Dicheva and Close, 1996; Kahney, 1982; Kurland and Pea, 1985; Wilcocks and Sanders, 1994). While there are many different models with many different names, they can be placed into one of four basic categories: the looping model, stack model (also called the copies model or the evaluation model), syntactic model (template, structural), and synthesis model (problem reduction, analysis, analysis/synthesis).

The looping model is an incorrect but common model of recursion and usually leads to confusion. This model stems from students' desire to represent the unfamiliar process of recursion in terms of the familiar process of iteration. The biggest problem with this model is that even though it is a faulty model of recursion, it does work well for simple cases of tail recursion. Unfortunately, such simple cases are often the first recursive problems studied, which allows students to be initially successful using this faulty model. However, the shortcomings of this model are soon apparent as students move on to problems with multiple base and/or recursive cases and embedded recursion. Kurland and Pea (1985) studied a group of seven children ages eleven to twelve as they attempted to predict and explain how several recursive Logo procedures worked. They concluded, "the most pervasive problem for all children was this tendency to view all forms of recursion as iteration" (p. 240).

The stack model is a correct, but limited model of recursion. This model is useful in tracing recursive programs and derives its name from the similarity to the way memory is allocated from the computer's stack as recursive calls are made. Kahney (1989) claimed that this is the model of recursion held by expert programmers

and tested this assumption with a group of nine expert programmers and thirty college students enrolled in a psychology course. The students' previous programming experience was not known. The subjects were given a problem to solve and three programs to solve it. The students were to choose which program would correctly solve the problem. One solution did not solve the problem and was not recursive. The other two solutions both solved the problem and both were recursive, one using tail recursion and the other embedded recursion. Kahney claimed that students who identified both correct solutions had demonstrated that they possessed the stack model of recursion. Those that identified the tail recursive program as correct but not the embedded recursive program were said to possess the looping model of recursion. Eight of nine experts did correctly identify both correct solutions while only one of the thirty students identified both correct solutions. It is important to note here that students and experts were considered to understand recursion if they identified the correct solutions from those presented. None of the students or experts were asked to write any recursive code.

Wilcocks and Sanders (1994) agreed with Kahney that the stack model is required to understand recursion and addressed the question of how to convey the model to the student. They attempted to do so using a program animator that graphically illustrated the execution of a recursive program. Specifically, a windowing environment was used to convey the stack model to the student. That is, as each recursive call was made, a new window with a copy of the code and the newly instantiated parameters would open on top of the previous windows. As the code in these windows finished executing, they were closed and the window containing the calling code would again be visible. However, they consider understanding of recursion to be the ability to explain how recursive code executes, in this case, in terms of their animator. There is again no mention of having students actually write recursive code.

Bhuiyan et al. (1989) discuss the stack model and note "the limitation of the stack model is that it explains nothing about how to derive the base cases and recursive cases. Therefore, it seems that there is limited use of this model in formulating a recursive solution. In our investigation, no one was found to directly use this model in formulating a recursive solution, but many used it extensively for tracing" (p. 138). They continued to discuss in this paper and in three subsequent papers (1991, 1992, 1994) two further models of recursion, the syntactic model and the synthesis model.

Students possessing the syntactic model of recursion have a set of recursive syntactic structures or templates that they use whenever they are writing a recursive program. Sophisticated syntactic models can have constructs for tail and embedded recursion as well as multiple base cases and multiple recursive cases. The biggest drawback to the syntactic model is that it often leads to trial and error methods of filling in the blanks in the template rather than a careful examination and subsequent understanding of the problem. Thus, students are often able to arrive at correct solutions using the syntactic model, but they frequently have little understanding of the recursive nature of the problems they have solved.

The synthesis model is the most general and abstract model of recursion (Bhuiyan et al., 1989). This model "gives the ability to reduce a given problem into smaller ones and to synthesize the corresponding solutions into a global solution for the problem" (Bhuiyan et al., 1991, p. 123). Problem solving with this model

requires identifying the base case(s) and recursive case(s) and most important, the relationship between the recursive and base cases that ensures successive recursive calls will eventually reach a base case. In other words, finding the pattern or structure in a problem that lends itself to a recursive solution.

Bhuiyan et al. (1994) claim that experts possess this synthesis model. Although there appears to be disagreement between Bhuiyan et al. and Kahney, who claimed experts possessed the stack model, it is likely both are right. Bhuiyan et al. claim that expert programmers use the synthesis model while formulating a solution for a problem, but notes that the synthesis model does not help directly in tracing a program. Since tracing and debugging skills are essential to the programming process, expert programmers probably possess both the stack model and the synthesis model of recursion. However, Anderson (1988) notes "it needs to be stressed that the major problems of novices are with initial program generation, and not with debugging." Also, in learning recursion and developing these models, it is certain that students will gain experience with numerous recursive syntactic structures, thus developing at least in part the syntactic model. In fact, Bhuiyan et al. use the idea that students will evolve from one model to another in building the programming environment described below. Thus, it appears that each of the stack, syntactic, and synthesis models of recursion are important, and each should be addressed when teaching recursion.

Programming Environments

Pirolli (1986) and Bhuiyan et al. (1994) have created the most notable programming environments designed to aid in learning recursive programming. Pirolli's LISP Tutor monitors and interacts with students as they write LISP programs. The tutor contains knowledge of the ideal solution strategy for each problem as well as commonly made errors. As a student writes a program, the tutor compares their progress to the ideal solution strategy and the known bugs. When the student's code is on a correct solution path, the tutor remains quiet. When the student appears to be on an incorrect path or a path that contains known bugs, the tutor provides feedback and hints to guide the student back onto the correct solution path.

Pirolli performed a study with twenty students enrolled in an introductory programming course to compare the efficacy of the LISP Tutor as compared to standard instruction. The students were divided into two groups matched on prior programming experience, SAT scores, and prior programming grades. Students in each group were given the same lectures and instruction booklets that "discuss how recursive functions have terminating cases, recursive cases, and recursive relations, illustrate how recursive functions operate, and present examples in the context of this discussion. The lesson booklets also outline the general procedure for planning the recursive cases of a function" (p. 337). Students in each group were then asked to solve a set of eighteen problems, one group using the LISP Tutor, the other using a standard LISP environment. The subjects in the tutored group were able to solve all of the problems while six of the ten non-tutored subjects were not. The tutored subjects averaged just less than six hours to complete the problems while the non-tutored group averaged slightly more than nine hours. Finally, on a fourteen-point test given at the end of the course, the tutored students scored significantly higher (7.60) than the non-tutored group (5.19).

All of this seems promising, but closer scrutiny reveals some areas of concern. First, the feedback and hints provided by the LISP Tutor should be examined. In describing the LISP Tutor, Pirolli says (p. 324):

> A basic premise in this article is that learning recursion can be facilitated by communicating the underlying goals and plans for programming recursion to students. To a large extent the goals and plans for programming recursion are determined by knowledge of the underlying structure and functionality of recursive programs. A crucial factor in learning to program recursion seems to be the knowledge that recursive functions can be characterized as having terminating cases, recursive cases, and recursive relations. A crucial skill seems to be the ability to plan out and identify the relation between the result of the function and the result of a recursive call to the function (the recursive relation).

This implies that the intent of the LISP Tutor is to help students develop the synthesis model of recursion. However, looking at a sample screen of the LISP Tutor reveals that it is based on the syntactic model. Students are provided with code templates for each function with tags such as '<BASE-CASE>' and '<RECURSIVE-CASE>' marking slots that are to be filled by the student. Hints include directions such as "Code the recursive case(s)" indicating what slot to attempt to fill next.

Second, when using the LISP Tutor, students can select a slot to be filled and are provided with a menu of choices to fill that slot. Recall that in a somewhat similar situation Kessler and Anderson (1986) found that students engaged in significant random behavior, or guessing. Pirolli did not address this issue, yet it is quite likely that the tutored group's higher success rate on the programming exercises can be attributed in large part to trial and error programming. Finally, recall the test scores for the tutored group (7.60) and non-tutored group (5.19) and that these scores were on a fourteen-point test. Even though the tutored group's scores were higher, their average score was barely above 50%. This is dramatically less than the 100% success rate achieved on the programming exercises, thus indicating a dependence on the LISP Tutor.

Bhuiyan et al. have used their work in characterizing students' mental models of recursion (1989, 1991) to design and build PETAL, a programming environment to support learning of recursion (1992, 1994) that utilizes different PETs, or Programming Environment Tools. Before describing PETAL, it is important to note the slightly different mental models used in its design. First, the stack model was not a consideration in their design because "our earlier studies have shown that learners seldom use a trace method to generate recursive solutions, and such an approach, if taken, usually results in incorrect solutions" (1994, p. 119). The models they describe and use in designing PETAL are the syntactic model, the analytic model, and the analysis/synthesis model. Their syntactic model is described as expected: a set of templates for different forms of recursion. The analytic and analysis/synthesis models are characterized slightly different than the synthesis model presented earlier. The analytic model "involves the ability to analyze the input-output requirements of a programming problem and to determine the input conditions and corresponding output actions together with associated transformations that will satisfy these requirements" (p. 118). The analysis/synthesis method is "the ability to reduce a given problem into smaller but similar ones and to synthesize the corresponding solutions to those smaller problems into a global solution for the problem" (p. 119). Examination of the PET derived from the analytic model shows it to be only a small step up, if any, from the syntactic model. The analysis/synthesis

model on the other hand implies a true understanding of and ability to identify the recursive structure of a problem.

The PETAL system contains three PETs, one for each of the syntactic, analytic, and analysis/synthesis models. When a student starts PETAL, they select a problem and code the function header. This is done in a single window that contains a list of thirty problems that can be selected, a text area showing a description of the selected problem, another text area showing example uses of the selected function, and two text editors that allow the student to enter the function name and the parameter list. The actual LISP code for the function header is then generated for them, removing the need to worry about syntax details such as parentheses. Next, the student chooses a PET and generates a solution for the problem using the PET. After generating the solution, the student can run their solution in a standard LISP environment.

The Syntactic PET is very much similar to Pirolli's LISP Tutor; students are shown a template and given a list of possible "code chunks" to fill in the empty slots. The Analytic PET is also surprisingly like the LISP Tutor. There are two stages to developing a solution with the Analytic PET. The first is to determine the different input cases and corresponding output strategies. This is done in a window that contains lists of problem specific input cases and output strategies. The student simply selects from each list to develop a "course-grained" solution to the problem. Then they move to the coding stage which is essentially a more problem specific Syntactic PET. That is, for each of the input cases and output strategies the student selected in the previous window, they are shown a template and a list of possible code chunks to fill it.

Based on the description of the analysis/synthesis mental model, the Analysis/Synthesis PET appears to hold the most promise. However, their very brief and vague description of the Analysis/Synthesis PET concludes with "the Analysis/Synthesis PET is designed for assisting LISP programmers with constructing solutions to very complex recursion problems. As such, it was not necessary for solving problems in the present study. Although it is briefly described for completeness, it was not evaluated in this study" (p. 125). The brief description of the PET says only "the Analysis/Synthesis PET provides only a few problem-specific natural language phrases. The learner breaks the problem into smaller but solvable sub-problems using these phrases" (p. 125). The reader is left to wonder what the phrases are and how the student might use them in breaking down a problem.

To evaluate the PETAL system a six-week short course in introductory LISP was offered to students enrolled in a first year Pascal programming course. Twelve volunteers enrolled in the course, but three of them eventually dropped out leaving five subjects in the experimental group using PETAL and four subjects in the control group using a traditional LISP editor. Based on high school grade averages and previous programming experience, the groups were considered more or less equal. During the first two weeks of the course, both groups were taught basic concepts of LISP such as arithmetic, logical primitives, list constructors, the conditional operator, and user defined functions. During this phase, both groups used a standard LISP environment for their problem solving. During the next three weeks students were taught recursion and attempted to solve a set of nine programming problems. The groups attended separate lectures and each student

attended individual labs using either the PETAL system (experimental group) or a standard LISP environment (control group). During the lab sessions a teaching assistant was available to provide limited assistance. "For example, the teaching assistant provided help when a learner had a misconception about PETAL or the LISP environment. He also provided help when a learner had misconceptions about the problem statement or LISP primitives as long as such misconceptions were not directly related to solving the problem" (p. 127). When a student reached a total impasse on a problem, the teaching assistant would provide more direct help, but "was cautious not to directly teach a learner about recursion or recursive problem solving. Furthermore, he was also cautious not to derail a learner from his or her current problem solving strategy through an intervention" (p. 127). Finally, a paper and pencil post-test was given during the last week of the course. This test was the same for both groups.

Their results showed that students using the PETAL system attempted and solved significantly more problems in significantly less time. The students using PETAL attempted, on average, 6.20 problems and correctly solved 5.00 problems, compared to 4.50 and 0.75 for the control group. This does not seem surprising considering the amount of assistance the PETAL system provides and the amount of assistance the teaching assistant did not provide. A more compelling result is that students in the experimental group scored significantly higher on the post-test. Students who used the PETAL system scored an average of 3.0 out of 4.0 possible on three recursive problems. All five of the students in this group correctly completed at least one problem and none of them scored 0 on any problem. Of the four students in the control group, none of them correctly solved any of the problems. In fact, the entire group scored 0 on all but two of the problems, and only scored 1 point on those problems. This demonstrates a total lack of understanding for this group.

As with the LISP Tutor, these results seem very promising. However, closer scrutiny reveals some unanswered questions here as well. First, there is no control over which PET is used. Their theory is that as students learn about recursion they progress through the different mental models, moving towards the most complete model, the analysis/synthesis model. But students are allowed to use any PET they desire, and in fact, two of the five students used the syntactic PET almost exclusively. Also, since the Analysis/Synthesis PET was only described by the authors and not used at all by the students, there is no way to determine if any students actually reached this level of understanding. Second, there is no mention made of discussing solutions to the programming problems that were attempted but not solved. This means that the students in the control group that did not solve any problems, or only solved at most one or two problems, saw very little if any correctly written recursive code. And they certainly did not have the opportunity to experiment with the correctly written recursive procedures over a three-week period, as did the students using the PETAL system. Finally, the exact problems used in the programming exercises and on the test are not discussed. Thus, if the problems on the test were the same or very similar to those used as programming exercises, then the students using the PETAL system would have an even further advantage. The complete failure of the control group to solve any of the test problems supports these deficiencies. No other researchers have had such complete failure by their control groups, indicating that the control group in this experiment was not given a fair chance at learning recursion.

Successful Simulations

Much research supports the idea that learning is dependent upon the learner's previous knowledge (Andre and Phye, 1986). As new ideas are presented, learners incorporate them into their existing knowledge structures while attempting to understand them. As they move towards understanding, their existing knowledge structures are modified as necessary to make the new idea fit. Without appropriate existing knowledge structures and prior experiences, students will have a difficult time gaining new knowledge and understanding. Thus, providing students with appropriate prerequisite knowledge is an important instructional goal.

Many researchers have claimed that there are naturally existing examples of recursion in our lives (Riordon, 1984, Troy and Early, 1992). Two examples are the series of reflected images visible when standing between two mirrors and the picture generated by pointing a video camera at its own monitor. While these examples may capture the basic idea that to be recursive, an object must somehow contain a copy of itself, they do little to help a novice programmer understand recursive programming.

To expect these examples to be sufficient prerequisite knowledge on which to base recursive program generation is quite unrealistic. Roberts (1986) agreed, noting that the difficulty for students in learning recursion is that "unlike other problem solving techniques which have closely related counterparts in everyday life, recursion is an unfamiliar idea and often requires thinking about problems in a new and unfamiliar way" (p. 1). That is, most students have followed directions such as a recipe for baking a cake. These instructions may have included a loop or even a procedure or function. However, few will have encountered a set of directions written recursively.

One approach to providing this prerequisite knowledge has proven successful. Specifically, simulations that put the learner in control of a manipulative model have been shown to help promote a deeper understanding of basic memory operations (Hooper and Thomas, 1990), looping (Upah and Thomas, 1993), and elementary genetics (Brandt, Hooper, and Sugrue, 1991). It is important that such simulations are manipulative in order to actively involve the learner rather than have them passively watch pre-programmed instruction.

Memops is a simulation of basic memory operations. Students are presented with a graphical representation of a five-element integer array labeled X and a single integer variable labeled Z. Commands to manipulate the model are entered at a prompt, one at a time. In the first set of exercises, students use the 'load' command to move the smallest value, move the largest value, swap two values, and sort the values into ascending and descending order. The 'load' command copies a value from one location into another location. When a 'load' command is entered, students see an animation of the value moving from its start location to its destination. After the animation, students can see that the value moved remains in the original location and there is a new a copy in the destination location. The value that was previously in the destination location is gone. For example, the command 'load x[2] with x[5]' would copy the value in location x[5] into location x[2]. Whatever value was originally in x[2] is gone.

In the second set of exercises, the values are hidden from view. Students must use the 'compare' command to determine relationships between values. When a 'compare' command is entered, students are shown a message that displays which of the two values being compared is smaller. The syntax of the 'compare' command is very similar to that of the 'load' command. For example, the command 'compare x[2] with x[5]' would compare the values in locations x[2] and x[5].

In Memops students use commands, which allows discussion of programming commands. They put commands together to solve problems, which allows discussion of algorithms. They manipulate the memory model, which allows discussion of memory and what happens inside the computer during execution. This leads to an understanding that writing to memory is a destructive operation and fosters development of the strategy of using temporary storage to preserve information.

Loops is a simulation of the looping control structure. Students are presented with a graphical representation of a coin hopper. When a 'getcoin' command is executed, the hopper dispenses a coin and places the numeric value of the coin (25, 10, 5, 1, and 0) in the variable 'newcoin'. Students are first presented with a short code segment that uses a while loop to calculate the sum of all the coins in the hopper (when the 0 coin is dispensed, the hopper is empty). The exercises are to modify the sample code so that it counts the coins, calculates the average of the coins, sums the coins up to but not including the first quarter, and sums the coins up to and including the first quarter. These tasks are designed to bring students into contact with each of the important ideas in looping. Specifically, the initialization, the boolean condition, the loop body, and the critical idea that the loop body must do something to affect the boolean condition.

It is important to remember that these simulations are not intended to teach students everything there is to know about memory operations or looping. Rather, they are intended to provide students with thought-provoking experiences that will make future classroom discussions and assignments more meaningful. This is accomplished by allowing them to manipulate important/critical ideas in understanding programming. Thomas and Boysen (1984) classify these as experiencing simulations. That is, they are used to "set the cognitive or affective stage for future learning" (p. 15). It is intended that students complete these simulations before formal instruction. Brant, Hooper and Sugrue (1991) performed a study which "supports the theory that the effectiveness of the simulation is dependent upon the sequence of presentation of learning activities to the students" (p. 477).

As mentioned above, Wilcocks and Sanders (1994) have developed a simulation of recursion. However, the very important idea that the simulation must be manipulative has been left out; their simulation only animated the execution of predetermined examples. More important is that their focus was on developing only the stack model, not the more robust synthesis model. Thus, their simulation did not fully address the critical features of recursion.

Critical Features of Recursion

The critical features of recursion are generally identified as the base case(s), the recursive case(s), the flow of control as recursive calls are made, and the notion that the recursive case must move in some way closer to the base case. An understanding of these features is necessary for a student to be able to read and understand recursive code. However, there is a much more fundamental feature of recursion that is often overlooked. In recursive problems and their solutions, there exists a pattern or relationship between each successively smaller recursive case and the base case as well as a relationship between the results of the smaller cases and the final solution. This is called the recurrence relation. It is this recurrence relation that allows a problem to be solved by a recursive solution.

Learning Style

Another consideration in the design of any instructional materials is that individual learners will respond differently to the same experiences. It is therefore prudent to examine how students with different learning styles might respond differently to a simulation of recursion.

Kolb's (1985) Learning Style Inventory identifies four different learning modes. Concrete experience emphasizes learning from specific experiences. Reflective observation involves viewing issues from different perspectives while looking for the meaning of things. Abstract conceptualization is characterized by logically analyzing ideas, systematic planning, and acting on intellectual understanding of a situation. Active experimentation involves risk-taking and experimentation.

From these learning modes, Kolb defines four learning styles. Accomodators combine concrete experience and active experimentation. People with this learning style have the ability to learn primarily from hands-on experience. Assimilators combine abstract conceptualization and reflective observation. People with this learning style are best at understanding a wide range of information and putting it in to concise, logical form. Convergers combine abstract conceptualization and active experimentation. People with this learning style are best at finding practical uses for ideas and theories. Divergers combine concrete experience and reflective observation. People with this learning style are best at viewing concrete situations from many different points of view. The question this theory raises is where simulations fit into this framework. Do simulations serve as concrete experiences, meet the needs of active experimenters, or facilitate abstracting? No research was found which answered this question.

Summary

With all of the work on developing lesson plans for teaching recursion, there is little devoted to providing students the insight necessary to detect the recursive structure of a problem and determine how that structure lends itself to a recursive solution. This is primarily due to the focus of most lesson plans on the stack and syntactic models of recursion. However, the problem begins with the first introduction to recursion. Invariably, lesson plans begin with numerous non-computer examples of recursion such as standing between

two mirrors. While these examples may capture the idea that to be recursive, a procedure or function must somehow contain a copy of itself, it is quite a leap from these examples to even simple recursive procedures in Logo. Also, many of the recursive examples presented are not used to their full potential in later instruction. For example, the tape recording of a song with multiple copies spliced into the middle could be used to introduce embedded recursion. Much more distressing about the lesson plans is the importance placed on non-*computer* examples of recursion when what are really needed are non-*programming* examples of recursion. To avoid using the computer to introduce the topic ties our most powerful hand behind our backs! To use the recursive song example, why not make the Logo turtle sing?

Research into mental models of recursion has demonstrated much promise. Understanding how a student is thinking about recursion is an important goal. More important is understanding how a student should be thinking about recursion. It is generally agreed that the synthesis model is the strongest and most abstract model of recursion. However, the stack model can not be totally ignored, as tracing and debugging are important skills. What is lacking in this area is research into how to help students construct these mental models, especially the synthesis model. Wilcocks and Sanders' program animator focuses only on the stack model. Pirolli's LISP Tutor and Bhuiyan's PETAL system both stop short of the synthesis model and focus instead on the syntactic model. Bhuiyan's PETs, for example, "carefully avoid imposing any extra syntax burden on learners. The structure of recursion, especially the structure of the final solution, is kept implicit in the user interface of the PETs" (Bhuiyan et al., 1994). To actually teach and develop the synthesis model, we must do more than hold students' hands as they work with the syntactic model. Further, research done to test programming environments such as the LISP Tutor and PETAL must ensure that all participants in the control group are exposed to an equal number of similar problems and solutions as those in the experimental group. Then, post-tests will be measuring the effects of the presentation, not simply experience gained.

Finally, and most important, let us decide what it means to understand recursion. Imagine a school where graduated students claim to be literate. How upsetting would it be to find out that these students could read but not write? Similarly, it can not be said that a student understands recursion when the student is capable of reading a recursive program and predicting the results but incapable of writing a recursive program. A student who has a good understanding of the flow of control as function calls are made will likely be able to read and understand recursive code, tracing the execution and predicting the results. However, there are no guarantees that this student will be able to generate recursive code to solve a problem. On the other hand, a student who is able to detect the recurrence relation in a problem and write recursive code to solve the problem will almost certainly be able to read and understand recursive code. Thus, it is imperative that the primary goal of teaching recursion be to develop this ability to think about and detect the recursive nature of a problem. From this point, the rest will follow.

SIMULATION

The stated goal of this research was to develop a computer simulation to aid in learning recursive programming. The software and associated lesson plan were intended to assist students in developing both the stack and synthesis models of recursion while avoiding other, faulty models. Students were implicitly guided towards an understanding of the stack model through the use of a windowing environment similar to that of Wilcocks and Sanders (1994). Students were more actively guided towards development of the synthesis model. The simulation and the tasks to be completed were designed to encourage students to think about the recursive structure of a problem as they attempted to solve it.

There are two distinct parts to this simulation. In both parts the students are presented with a two-dimensional array of squares. Some of the squares are filled to produce a goal pattern. Students are also presented with an array of squares that is initially empty, called the current pattern (Figure 1). Their task is to fill squares in the current pattern to produce a copy of the goal pattern. Two commands are available for use, 'view' and 'fill'. As its name implies, the 'fill' command is used to fill squares. When a 'fill' command is entered, all squares visible in the current pattern are filled. The 'view' command is used to change the portion of the current pattern that is visible. When executed, the 'view' command opens a new window with only the indicated portion of the current pattern visible. To indicate which quadrant of the current pattern to display in the new window, the student must specify upper left, upper right, lower left, or lower right. While using the simulation, details about each command are available in a help menu.

Command Lesson

The first part of the simulation, the Command Lesson, is an experiencing simulation and is used before formal instruction on recursion. In this part, the student manipulates the current pattern by entering commands individually in each window. Commands can only be entered in the most recently opened window. The main window is shown in Figure 1. This window is opened when the student begins the simulation. It contains the goal pattern that the student is trying to create and the current pattern, which is initially empty.

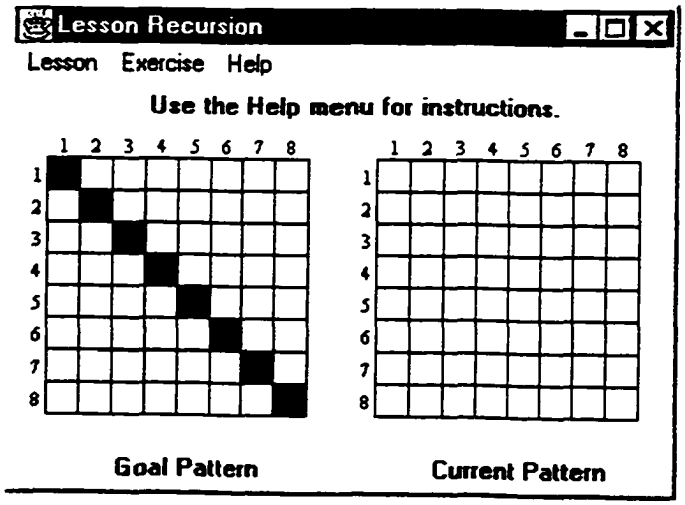


Figure 1. The main command window.

To begin entering commands and manipulating the current pattern, the student selects the Begin option from the Exercise menu. This will open the command window, shown in Figure 2. Commands typed in the command editor are applied to the entire pattern shown in the window. Thus, a 'fill' command entered in the command window in Figure 2 would fill the entire display. Since this is not the desired goal pattern, the student must first use the 'view' command to change which squares are shown in the window.

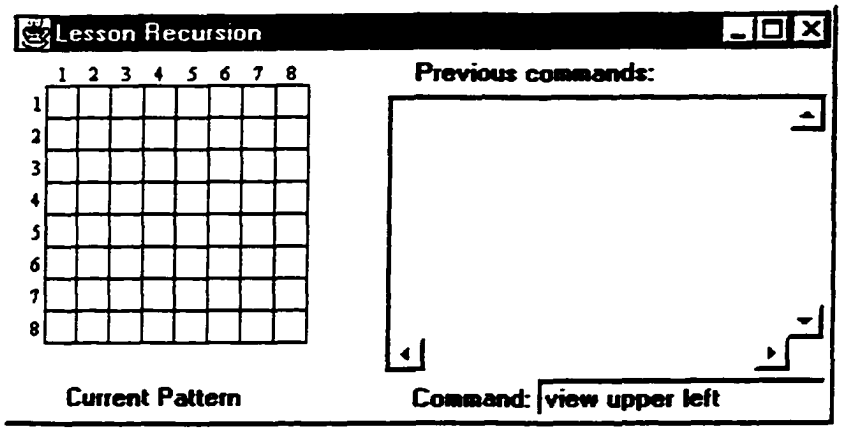


Figure 2. The command window.

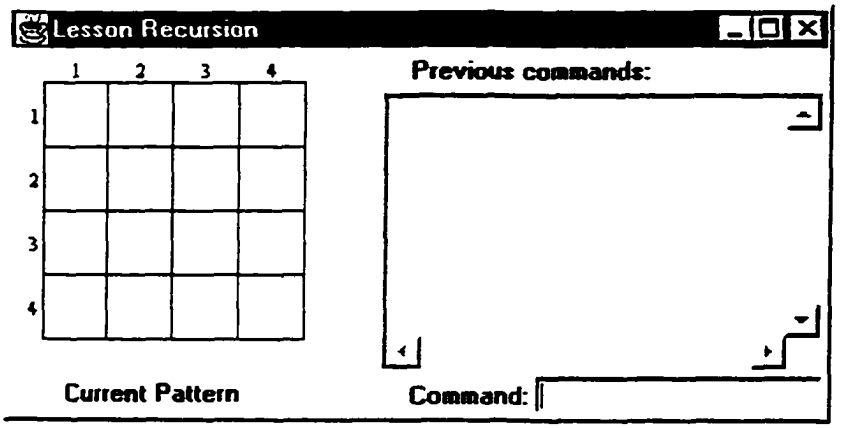


Figure 3. Command window after executing 'view upper left'.

The command 'view upper left' has already been typed in the command window in Figure 2. When the student presses the enter key after typing this command, another command window will open with only the upper left 4x4 square from the original 8x8 pattern visible (Figure 3).

Note that at this point there are three windows open. The main window in Figure 1 is open and is always visible so that the student can see their progress (the current pattern in this window is updated whenever 'fill' commands are entered). The command window in Figure 2 is also open. However, when the window in Figure 3 is opened as a result of entering the 'view upper left' command in Figure 2, it is opened slightly offset,

but mostly covering the window in Figure 2. Commands can not be entered in the window in Figure 2 as long as there are windows on top of it. This use of the windowing environment emulates the stack model of recursion and clearly models the flow of control as recursive calls are made.

To complete this exercise, the student would enter another 'view upper left' command in the command window in Figure 3. This would open yet another command window with only the upper left four squares visible. Another 'view upper left' command in that window would finally open a command window with only the single square in the upper left corner visible. At this point a 'fill' command would be entered to fill this square. Then a 'return' command would close this window and return control to the previous window, the one with four squares visible. At this point, the student must enter a 'view lower right' command in order to view and then fill the square in row 2, column 2 of the original display. This strategy would continue until the student had completed the pattern.

The other component of the command window is a list of previous commands used in that view. After finishing each exercise, the student will be asked to reflect on what commands were in this list in each window just before the 'return' command was entered and the window was closed. The insight to be gained is that the list of commands in each window should be exactly the same except for the windows where the 'fill' command was entered. This facilitates discussion of the base case and the recursive case of a recursive program. That is, the 'view' command is essentially a recursive call and must be given in all windows with more than one square visible. The 'fill' command is a base case and is entered in windows with only one square visible.

Code Lesson

The second part of the simulation, the Code Lesson, promotes generalization and abstraction from the first part of the simulation. The main code window is shown in Figure 4.

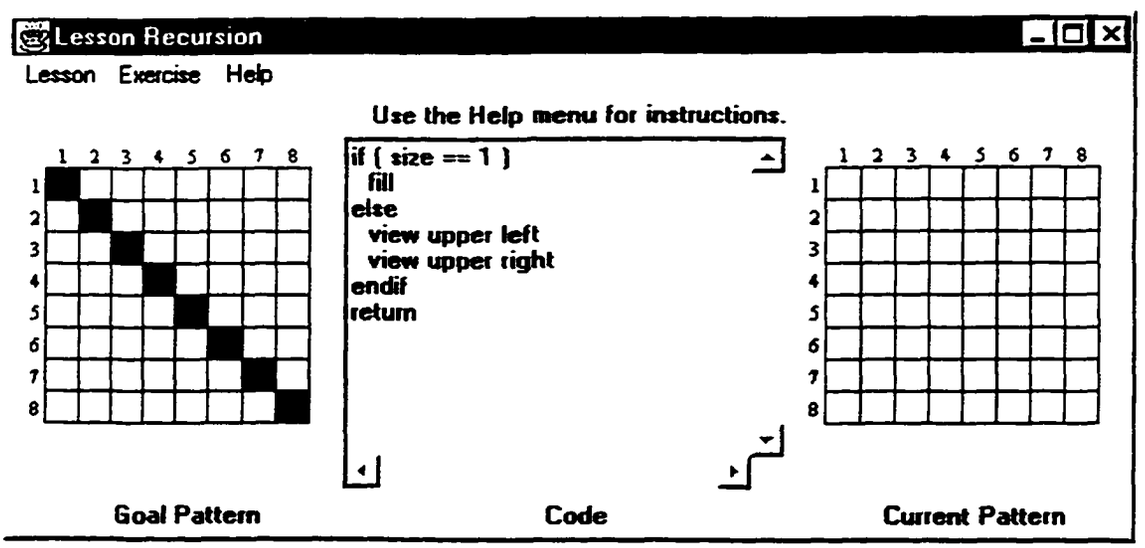


Figure 4. The main code window.

In the main code window students again see the goal pattern and the current pattern. These function in the same manner as those in the main command window. However, there is also an editor where students can enter some code. This code is executed in each successive window as it opens. The code in the main command window shown in Figure 4 would correctly produce the diagonal goal pattern.

When the student selects the Begin option from the Exercise menu in the main code window, the code window shown in Figure 5 is opened. Again, this window is opened so that the main code window is still visible which allows the student to track the progress in the current pattern. Subsequent code windows will be opened slightly offset but mainly covering the previous code window, as with the command windows. When a new window opens, previously opened windows are unavailable until the current window is closed.

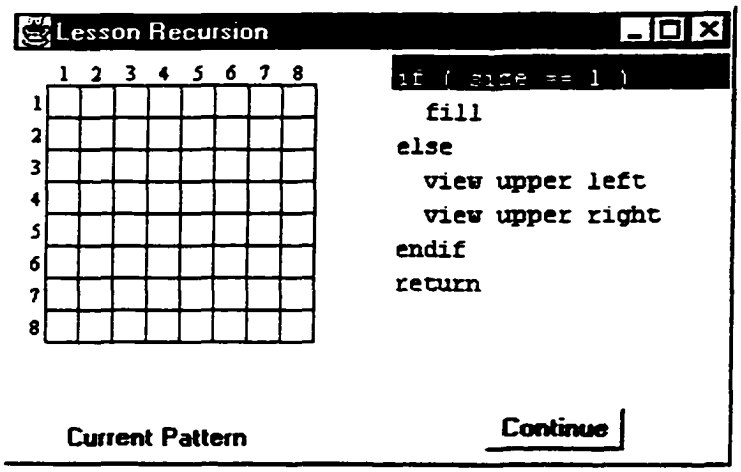


Figure 5. The code window.

The code window shows the current pattern and the code that is to be executed. This window acts similar to a debugger in that the current line of code is shown in inverse video. This line of code will be executed when the user clicks on the Continue button. When a 'view' command is executed, another window with the same code and the specified view is opened. Creating code that will execute the proper instructions for each view regardless of size forces the students to generalize their solutions. More importantly, to be able to enter the appropriate commands in the Command Lesson and to generalize those commands into the appropriate code in the Code Lesson, students will need to first search for the recursive structure of the pattern.

It is very important to note that this discovery of the recursive pattern, or the possible insights into the flow of control, the base case, and the recursive case will not happen automatically as a student uses the simulation. Rather, as an experiencing simulation, the intent is to set the stage for future classroom discussions. This discussion of the simulation and how it exposes the critical features of recursion is perhaps the most important part of the lesson. The complete lesson plan will be presented in the following chapter.

Exercises

The most important step in designing the simulation is the selection and sequencing of exercises so that the learner will be guided toward a confrontation with the critical features of recursion. There are six exercises in the Command Lesson. The first exercise (Figure 6) is to fill the single square in the upper right corner. This is solved by entering three 'view upper right' commands to bring the single square into view, and then entering a 'fill' command to fill the square. However, before the students can move on to the next exercise they are asked to close all windows that remain open. That is, since each of the three 'view' commands opened a new window, there will be a total of four windows open. Each of these must be closed to complete the exercise. This introduces flow of control and illustrates that when a procedure makes a recursive call, it suspends its own execution and waits for the called procedure to finish.

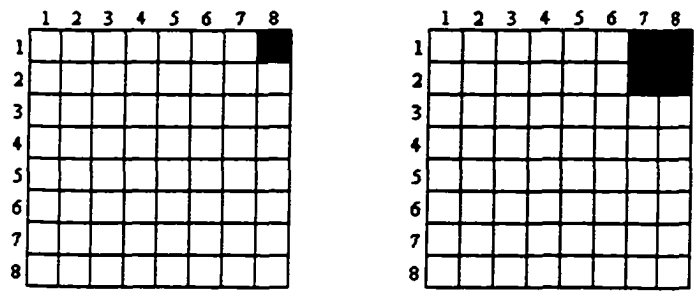


Figure 6. Command lesson exercises 1 and 2.

The second exercise (Figure 6) is to fill the four squares in the upper right corner. The only difference from the first exercise is the base case. This exercise reinforces the flow of control and provides an opportunity to think about and discuss the difference between the base case and the recursive case.

The third exercise (Figure 7) is to fill the single squares along the diagonal from the upper left corner to the lower right corner. As a students works through the windows bringing the proper squares into view and filling them, the ideas of flow of control, recursive cases, and base cases are again illustrated. Also, the student must deal with embedded recursion since each window that does not contain a base case requires one 'view' command to complete the upper left section and another to complete the lower right section.

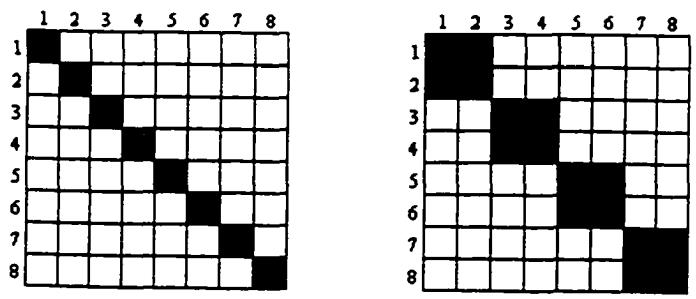


Figure 7. Command lesson exercises 3 and 4.

The fourth exercise (Figure 7) is to fill the same diagonal with squares of size four. The only difference between the two exercises is the base case. The fifth and sixth exercises (Figure 8) are very similar to the third and fourth, dealing instead with the diagonal from the upper right to the lower left corner. These exercises are intended to provide some practice with the concepts the students have encountered in the first four exercises. This very simple set of exercises provides the instructor the opportunity to discuss all of the critical features of recursion in a meaningful context.

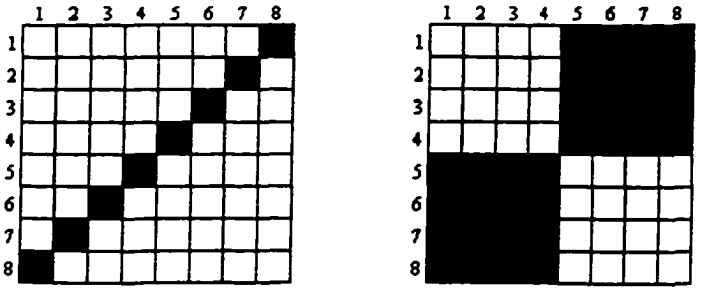


Figure 8. Command lesson exercises 5 and 6.

The Code Lesson contains eight exercises. The first four exercises use the same patterns as the first four exercises in the Command Lesson (Figures 6 and 7). The next three exercises (Figure 9) are similar to the first four, but they use patterns that the students have not yet seen. Finally, the last exercise (Figure 9) is different from the previous exercises in that it requires students to identify two different recursive cases.

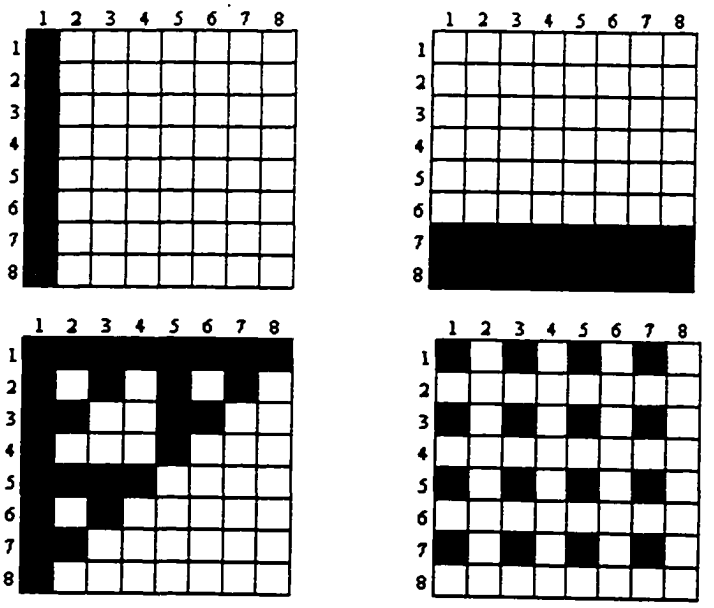


Figure 9. Code lesson exercises 5 through 8.

```
if( size = 1 )
  fill
else
  view upper left
  view upper right
  view lower left
endif
```

```
if( size = 1 )
  fill
else if( size = 4 )
  view upper left
else
  view upper left
  view upper right
  view lower left
  view lower right
endif
```

Figure 10. Solutions to exercises seven and eight.

For comparison, consider the solutions to exercises seven and eight (Figure 10). Notice the need for two conditional statements in the solution to the last exercise. This aspect of the solution is different from previous exercises and forces the student to modify and extend their understanding of recursion.

Summary

There are two distinct parts to this simulation. In the first part (Command Lesson) students create the patterns by entering individual commands in each view. To successfully solve the problems students must determine how commands entered in each sub-view will contribute to the overall pattern. In the second part (Code Lesson) students write one short code segment that will be executed in each view. To successfully solve the problems students must make generalizations from the individual commands entered in the first part of the lesson. Both parts of the simulation explicitly promote development of the synthesis model of recursion by encouraging students to identify the recursive structure of the patterns and implicitly promote development of the stack model of recursion through the use of the windowing environment.

METHODS

Subjects

The subjects for this research were students enrolled in the fall, 1997, introductory computer programming course at Simpson College in Indianola, Iowa. These students were primarily in their first and second years of college. There were two separate sections of the course (A & B), both taught by the primary researcher. Each section had three one-hour classroom meetings and one two-hour lab meeting per week. Section A (N = 25) was the experimental group and section B (N = 24) was the control group, as determined by a coin toss. The subjects were required to participate in the study, as recursion is an important topic to be covered in the course. However, they were asked to sign an informed consent form (Appendix D) and were given the opportunity to withhold their scores from analysis.

Instruments

There were four major sources from which data pertinent to the research questions was collected. First, Kolb's (1985) Learning Style Inventory was administered so that comparisons could be made between learning style and achievement. Second, ACT scores and overall course percentage (previous exams, quizzes, and programming assignments) were used to determine that the experimental and control groups were equivalent. Third, a record of on-line programming actions on a set of exercises was kept, including source code for each attempted run, total number of runs, and number of correct solutions generated. This information has been termed a programming "protocol" and has been shown to be an effective method of evaluating student performance on programming problems (Hooper, 1986). Finally, several paper and pencil exams were given. First was a quiz administered during the final class period of the study (Appendix A). Next, several questions on recursion were included on the final exam (Appendix B). And last, a follow-up exam was given during a class period of the second semester programming course the following spring (15 of 25 students from the experimental group and 10 of 24 students from the control group enrolled in the second semester course). It is important to note that all grading was done based on understanding of concepts rather than syntax. (Appendices A, B, and C contain grading criteria as well as the exam questions.)

Analysis

To ensure that the groups did not differ significantly in ability, t-tests were performed on their composite ACT scores and total course percentage. This course percentage included two exams, eight quizzes, and ten programming assignments. T-tests were also used to compare the number of exercises completed by students in each group, the number of attempts required to complete the exercises, scores on the final exam, and scores on a follow-up exam administered to those students who enrolled in the second semester course. Finally, an analysis of variance was performed to determine if student learning style affected performance.

Research Design

The research design (Table 1) used is best described as a quasi-experimental design because it was not possible to randomly assign the subjects to the control and experimental groups. The primary independent variable was the method of introducing recursion used with each group. Recall that there were two distinct parts to the recursion simulation. In the Command Lesson students entered commands individually, while in the Code Lesson students entered complete code segments to solve the problems. The experimental group used the Command Lesson as their first exposure to recursion. The control group was presented a more traditional introduction to recursion based on their textbook. After these separate introductions, both groups used the Code Lesson simulation to complete a set of exercises. The data on number of exercises completed and number of attempts required to complete the exercises was taken from the exercise set used in the Code Lesson.

Table 1. Research Design

Experimental Group	LSI	S1	S2	R	Q	T	F
Control Group	LSI	C1	C2	R	Q	T	F

LSI	Administration of Learning Style Inventory.
S1	Manipulative simulation lab experience (Command Lesson described above).
S2	Discussion of the simulation and identification of the critical features of recursion in terms of the exercises encountered in the simulation.
C1	First classroom lecture on recursion. This was a traditional introduction to recursion. A definition of recursion was presented along with numerous examples.
C2	Second classroom lecture on recursion. The critical features of recursion were identified and discussed in terms of the examples presented in the previous lecture.
R	Recursive coding simulation. (Code Lesson described above.)
Q	Paper and pencil quiz administered during final class period.
T	Paper and pencil exam administered during final exam period.
F	Follow-up exam administered during class period the following semester.

The study was conducted during the last week of the semester. During Monday's class, the experimental group met in the computer lab and worked with the Command Lesson simulation while the control group met in the classroom and received a more traditional introduction to recursion (similar to those described in chapter 2). On Wednesday, the experimental group met in the classroom and began to formalize their understanding of recursion based on their experiences with the simulation. The control group continued with their traditional discussion of recursion. Also, a brief introduction to the Code Lesson simulation they would be using in Thursday's lab was necessary since this group did not work with the Command Lesson. On Thursday,

both groups met in the lab and worked with the Code Lesson simulation. Finally, on Friday both groups met in the classroom and discussed their experiences with the Code Lesson. Then, the traditional Towers of Hanoi problem was introduced and students took a short quiz over this problem. The research design is summarized in Table 1 and more detailed lesson plans follow.

Experimental Group Lesson Plan

The experimental group met in the lab on Monday and was asked to complete the exercises in the Command Lesson. Upon arrival in the lab, students were greeted with the following comments from the instructor:

> The topic of the next few classes is recursion. Briefly, a function that contains a call to itself is said to be recursive. We will discuss this in more detail later. Today, we are going to use a simulation similar in design and purpose to the Memops and Loops simulations used earlier in the semester. That is, it is not intended to teach you everything there is to know about recursion; rather, the intent is to get you started thinking about recursion and to provide groundwork for future discussions.

Students were then given the remainder of the hour to work on the exercises. As they began to finish the exercises they were asked to write down the commands that were listed in each window immediately before they entered the 'return' command which closed the window. They were asked to think about differences and similarities between the lists of commands and bring these to class on Wednesday. The goal was to have students notice the difference between patterns requiring 'fill' commands and patterns requiring 'view' commands. Also, it was hoped that students would notice that in all of the cases that required 'view' commands, the same 'view' commands were used. These observations would serve as an introduction to the ideas of base and recursive cases.

On Wednesday students were asked to look at their lists of commands and discuss any similarities they had noticed. In discussing these similarities students were introduced to the base case, the recursive case, and the condition that recursive cases must move closer to a base case. Also, the flow of control was discussed as students were asked to describe what happens to each window as they enter 'view' commands and subsequent 'return' commands. During this discussion students were asked to think about how they determined what commands needed to be entered to produce the goal pattern. The plan was to have students think about the overall pattern of filled squares. Students were not yet expected to realize that determining this pattern is the most important part of generating a recursive solution. Rather, they were merely expected to realize that there is indeed a pattern and it does influence the commands they need to execute.

Next, students were asked if they could combine their lists of commands from all of the different windows into one set of instructions that would work if used in each window. The immediate response was that this could not be done since all windows did not need to execute the 'fill' command. They were then asked if they had any way of selectively executing an instruction. At this point students were introduced to the variable 'size' that could be used in a conditional statement to determine the number of squares in the current window. Through discussion and suggestions by various class members, a solution to the first exercise was generated and

written on the board. Now, students were asked to look back at the pattern they had observed when they were entering commands individually. It was suggested that this pattern would help them make a general list of instructions. This is the point where students were pushed to make the critical connection between the pattern of filled squares and the code segment that produces the pattern. Students were told that they would be doing more of this type of exercise the next day.

During the lab session on Thursday students worked with the Code Lesson. As they worked through the first four exercises with the same patterns as those in the Command Lesson they were encouraged to think about how they solved each exercise in the Command Lesson and remember the lists of commands that were in each window just before it closed. The challenge came when students moved on to exercises with new patterns. As students worked through these exercises they were told to think about how they would determine what commands to type if they were using the Command Lesson. The plan was again to have students think about the entire pattern of filled squares rather than a single square or smaller portion of the pattern. In addition to recognizing the pattern, they now had to generalize the commands they would use to create a single recursive code segment that would generate the pattern.

During the first half of Friday's class period, exercises from the Code Lesson were reviewed. At this point in the discussion, the recursive solutions were carefully presented and discussed using the synthesis model. As each problem was presented, the first question posed to students was to identify the pattern of filled squares that would allow a recursive solution. After generating a solution, students were encouraged to look at how that solution created the correct pattern in the original 8x8 square. For example, in the exercise with only the diagonal squares filled, students were asked to think of each recursive 'view' command as a single command that simply performed the required operations on the designated portion of the pattern. In other words, they were asked to think of the recursive calls abstractly, as a completed task, rather than as an even longer list of instructions to be executed. For some students there were still doubts, so the stack model was used to trace the solution and verify its correctness.

During the second half of Friday's class, students were introduced to the Towers of Hanoi problem. In this problem there are three towers, or pegs, onto which a stack of discs may be placed. Each of the discs is a different size and in the initial state of the problem, all discs are stacked on the leftmost peg with the largest disc on bottom and the smallest disc on top. The goal is to move all of the discs from the leftmost peg to the rightmost peg while adhering to the following two rules: (1) only one disc may be moved at a time; (2) at no time can a larger disc be placed on top of a smaller disc.

Students were first presented the problem with three coins (a nickel, a penny, and a dime) placed on the overhead projector. After solving the problem, students were asked to write down the required moves. Next, a fourth coin was added (a quarter). Again, students solved the problem and were asked to write down the necessary moves. Now, students were asked to compare the lists of moves required to solve the problem with three coins and four coins and look for a pattern. With some hints, students eventually noticed that the seven steps required to solve the three-coin problem were identical to the first seven steps of the four-coin problem

with only the spare and destination pegs reversed. Also, the seven steps required for the three-coin problem were identical to the last seven steps of the four-coin problem, again with only the peg names changed. These insights were used to generate the following algorithm:

> To move four coins from the start peg to the destination peg:
> Move three coins from the start peg to the spare peg,
> using the destination peg as a spare;
> Move the fourth coin from the start peg to the destination peg;
> Move three coins from the spare peg to the destination peg,
> using the start peg as a spare.

To end the class period, students were given a short quiz with a partially completed Towers of Hanoi program (Appendix A) and were asked to complete the program.

Control Group Lesson Plan

The control group met in the classroom on Monday and was given a more traditional introduction to recursion that followed closely the discussion and examples provided in their textbook. The introduction and definition of recursion was very short: "Recursion is an extremely powerful programming technique that allows you to solve very complex problems with short, simple programs. A function is recursive if it contains a call to itself." Next, the mathematical definition of the factorial function was written on the board:

```
1! = 1;
N! = N * (N-1)!
```

It was then demonstrated how this defines 4! (students were warned not to confuse the mathematical notation for factorial with the C++ notation for negation, both of which are the exclamation point.)Next, students were presented with a C++ definition of the factorial function:

```
int factorial( int n )
{ if( n == 1 )
  { return 1;
  }
  else
  { return n * factorial( n-1 );
  }
}
```

This was then traced using the stack model as it calculated `factorial( 4 )`. Careful emphasis was placed on the flow of control as each recursive call was made and as each recursively called function terminated. Finally, after presenting and discussing these examples, students were introduced to the critical features of recursion through a series of questions: (1) What is the first (base) case? (2) What is the n^{th} (recursive) case? (3) How is the n^{th} case related to the $(n-1)^{th}$ case? (4) What is it about the structure of the definition of the factorial function that allows it to be solved recursively?

Wednesday's class began with a review of the critical features of recursion as defined in terms of the factorial function and students were asked which of the features they felt was most important. After some discussion it was proposed that the most important aspect of writing recursive functions was to identify the

recursive structure of the problem. This was used as an introduction to the Code Lesson lab activity to be used in Thursday's lab. Since the control group did not use the Command Lesson, Wednesday's class finished with an introduction to the Code Lesson simulation. Students were first introduced to the 8x8 array of squares and the goal pattern with only the upper rightmost square filled. The 'view' and 'fill' commands were explained and it was demonstrated how these would be used to create this goal pattern. Next, the goal pattern with only the diagonal squares filled was presented and commands to create it were discussed. After discussing individual commands, generalized code segments to create these two patterns were presented and discussed using the stack model to trace their execution. At this point it was again emphasized that in order to generate the code, students had to first identify the pattern, then determine the base case and the recursive cases, and finally write the code. The deeper ideas of the synthesis model and abstraction were left for future discussion.

During Thursday's lab and Friday's class the control group participated in the same activities and discussions as the experimental group.

Summary

The independent variable in this research was the initial exposure to recursion received by each group. The experimental group used the Command Lesson simulation as their first exposure to recursion while the control group received a lecture-based introduction to recursion. The dependent variables were performance on a set of eight recursive programming exercises completed using the Code Lesson simulation, scores on recursive problems on the final exam, and scores on a follow-up exam. ACT scores and course percentages at the time of the treatment were used to test equality of the control and experimental groups.

RESULTS

In the first part of this chapter the results of the study are reported for each hypothesis. In the second part of the chapter these results are discussed. For the reader's convenience, tables of data and test results are included in the chapter.

The purpose of this research was to develop a computer simulation to aid in learning recursive programming. A quasi-experimental research design was used to evaluate the simulation and associated lesson plan. The subjects of the study were students enrolled in an introductory, college-level programming course. Two sections of the course, taught by the same instructor, were designated either experimental or control group by the toss of a coin. The control group consisted of 24 students and the experimental group had 25 students. The experimental group used the Command Lesson simulation as their first exposure to recursion while the control group used a more traditional, lecture-based introduction to recursion. Then, both groups used the Code Lesson simulation to complete a set of eight recursive programming exercises. The eight exercises fit into three distinct categories. The first four exercises were recall exercises as they used the same patterns as the first four exercises in the simulation. These patterns were also used as examples in control group lectures. The next three exercises were application exercises as they used different patterns, but required the same type of solution as the recall exercises. Solutions to the recall and application exercises required one base case and one recursive case. The last exercise was a transfer exercise as it required students to extend what they had learned to a new situation. Specifically, the solution required two recursive cases and one base case.

Group Comparison

One concern for this study was the limitation that the students could not be paired and randomly assigned to groups. To test for significant differences in overall ability between the two groups, t-statistics were computed using ACT scores and overall course percentage at the time of the treatment. There was no significant difference in ACT scores ($p < 0.47$) or course percentage ($p < 0.30$). These data, shown in Table 2, support the position that the groups were comparable.

Table 2. Group comparison.

Group	N	ACT Mean	ACT S.D.	Course Mean	Course S.D.
Experimental	25	24.40	3.69	81.74	9.11
Control	24	23.38	3.73	77.36	10.14

Completed Exercises

Hypothesis 1: Students in the experimental group will successfully complete more exercises. A t-test was used to test hypothesis one. There was no significant difference ($p < 0.35$) in the number of exercises completed by each group. The mean for the experimental group was 7.84 whereas the mean for the control group was 7.75. Inspection of individual scores reveal that of the 49 students that participated in the study, only

Table 3. Completed exercises.

Group	Mean	S.D.
Experimental	7.84	0.62
Control	7.75	1.03

four students (two from each group) did not complete all eight exercises. Of those four students, two (one from each group) were able to complete all of the recall and application exercises, failing only to complete the transfer exercise. The data is shown in Table 3.

Exercise Attempts

Hypothesis 2: Students in the experimental group will successfully complete all exercises in fewer attempts. The mean number of attempts required by students in the experimental group to complete all eight exercises was 18.28. The mean number of attempts required by the control group was 22.54. A t-test determined the difference was significant ($p < 0.03$). The experimental group required fewer attempts to complete the set of eight exercises. The attempts required to complete the subsets of exercises were also compared.

Hypothesis 2A: Students in the experimental group will successfully complete the recall exercises in fewer attempts. The mean number of attempts required by students in the experimental group to complete the recall exercises was 6.88. The mean number of attempts required by the control group was 6.92. A t-test did not reveal a significant difference ($p < 0.49$).

Hypothesis 2B: Students in the experimental group will successfully complete the application exercises in fewer attempts. The mean number of attempts required by students in the experimental group to complete the application exercises was 4.40. The mean number of attempts required by the control group was 5.13. Again, a t-test did not reveal a significant difference ($p < 0.11$).

Hypothesis 2C: Students in the experimental group will successfully complete the transfer exercise in fewer attempts. The mean number of attempts required by students in the experimental group to complete the transfer exercise was 7.00. The mean number of attempts required by the control group was 10.50. A t-test determined the difference was significant ($p < 0.02$). The experimental group required fewer attempts. These results are summarized in Table 4.

Table 4. Exercise attempts.

Group	Total Mean	Total S.D.	Recall Mean	Recall S. D.	Application Mean	Application S.D.	Transfer Mean	Transfer S.D.
Experimental	18.28	7.08	6.88	3.92	4.40	1.53	7.00	5.59
Control	22.54	7.17	6.92	4.10	5.13	2.44	10.50	6.13

Exam Scores

Hypothesis 3: Students in the experimental group will score higher on recursive programming questions on the final exam. (Appendices A, B, and C contain the exam questions and grading criteria.) The final exam was administered in two parts. The first part was a question on the Towers of Hanoi problem that was given during the final class period of the study. In this question students were asked to complete a partially written Towers of Hanoi program (Appendix A). The second part of the final exam was given during the final exam period and included four questions similar to those in the simulation and one question that asked students to trace the output of a completed Towers of Hanoi program (Appendix B). There was not a significant difference ($p < 0.11$) in the final exam scores. The mean score for the experimental group was 20.08 (out of 25 points possible) and the mean score for the control group was 19.29. These results are summarized in Table 5.

Table 5. Final exam scores.

Group	Mean	S.D.
Experimental	20.08	2.33
Control	19.29	2.01

Hypothesis 4: Students in the experimental group will score higher on recursive programming problems on a follow-up exam given the following semester. Fifteen out of twenty-five students in the experimental group and ten out of twenty-four students in the control group were enrolled in the second semester course and were available to take the follow-up exam. The limitation that students were not randomly assigned to groups was again a concern. To test for significant differences in overall ability of the subgroups of students available to take the follow-up exam, t-statistics were computed using ACT scores and overall course percentage at the time of the experiment. The results were almost identical to the original group comparisons, showing no significant difference in ACT scores ($p < 0.46$) or course percentage ($p < 0.31$). Thus, the subgroups were assumed to be equivalent. This data is shown in Table 6.

Table 6. Group comparison for follow-up exam.

Group	N	ACT Mean	ACT S.D.	Course Mean	Course S.D.
Experimental	15	24.13	3.93	84.20	7.05
Control	10	24.30	4.57	82.63	8.24

The mean score for students in the experimental group who took the follow-up exam was 10.80 out of 15 possible. The mean score for students in the control group who took the follow-up exam was 8.40. A t-test determined this was a significant difference ($p < 0.04$). The experimental group's scores on the follow-up exam were higher. These results are summarized in Table 7.

Table 7. Follow-up exam scores.

Group	Mean	S.D.
Experimental	10.80	3.00
Control	8.40	3.66

Problem Solving Strategies

Hypothesis 5: Students in the experimental group will use different strategies when solving recursive programming problems. Programming protocols were saved as each student worked through the exercises. The protocols contained information such as the number of attempts for each problem and the source code for each attempt. The protocols were examined in an attempt to identify and classify commonly used strategies. This hypothesis was originally included based upon the success of such programming protocols in studies on memory operations (Hooper and Thomas, 1990) and looping (Upah and Thomas, 1993). Unfortunately, programming protocol data in this study was not as useful as in previous studies. There were two primary reasons for this. First, the high success rate of the students made the protocols very short. Second, as discussed below, the similarity of the patterns and solutions for the basic 8x8 grid did not provide enough flexibility for the students to demonstrate different ways of thinking. Therefore, no conclusions about problem solving strategies could be reached based upon the programming protocol data.

Learning Styles

The Learning Style Inventory classifies learners as accommodators, assimilators, convergers, or divergers (Kolb, 1985). Of the twenty-five students in the experimental group, 6 were classified as accommodators, 10 as assimilators, 7 as convergers, and only 2 as divergers. Of the twenty-four students in the control group, 7 were classified as accommodators, 11 as assimilators, 4 as convergers, and again only two as divergers. Due to the very small number of divergers, they were excluded from this analysis. These totals are summarized in Table 8.

Table 8. Group learning styles.

Group	Accommodators	Assimilators	Convergers	Divergers
Experimental	6	10	7	2
Control	7	11	4	2

Hypothesis 6: Learning style will not affect the number of completed exercises. Only two students from each group were not able to complete all eight exercises. In both the experimental group and control group, one of these students was an accommodator and the other was an assimilator. Thus, it is clear that learning style did not affect the number of completed exercises.

Hypothesis 7: Learning style will not affect the number of attempts required to complete the exercises. A two-way analysis of variance was calculated to test this hypothesis. The analysis of variance did not indicate any significant difference due to group ($p < 0.06$) or learning style ($p < 0.71$). Also, no significant interactions were revealed ($p < 0.62$). The results are summarized in Table 9.

Table 9. Analysis of variance data for the number of attempts.

Source	DF	SS	MS	F-Value	P <
Group	1	216.78	216.78	3.61	0.06
Learning Style	2	42.24	21.12	0.35	0.71
Interaction	2	58.28	29.14	0.49	0.62
Residual	39	2343.96	60.10		

Hypothesis 8: Learning style will not affect student performance on exam questions. A two-way analysis of variance was calculated to test this hypothesis. The analysis of variance did not indicate any significant difference due to group ($p < 0.40$) or learning style ($p < 0.51$). Also, no significant interactions were revealed ($p < 0.09$). The results are summarized in Table 10.

Table 10. Analysis of variance data for the exam questions.

Source	DF	SS	MS	F-Value	P <
Group	1	2.97	2.97	0.71	0.40
Learning Style	2	5.72	2.86	0.69	0.51
Interaction	2	21.23	10.61	2.55	0.09
Residual	39	162.63	4.17		

Discussion

The nature of the problems being solved probably contributed to the high level of success. Using only the 8x8 grid of squares, there are a limited number of recursive patterns that can be created, and the code segments to generate these patterns are quite similar. The number of different patterns that can be generated by code segments with only one base case and one recursive case is even smaller. Both the recall and application exercises required only one base case and one recursive case. Also, as noted in the lesson plan, both groups had discussed solutions to the first two problems in class prior to working on the exercises. Thus, it is not surprising all but one student in each group completed all of the recall and application exercises. The transfer exercise was different from the rest, as it required two recursive cases. Even so, all but two students in each group were able to complete it.

The differences between the two groups appeared in the number of attempts required to complete the problems. There was a significant difference in the total number of attempts required to solve all eight exercises, but looking at differences in the smaller subsets of exercises (recall, application, and transfer) allows

more interesting conclusions. Since the recall exercises used patterns from the Command Lesson, it was expected that experience would enable the experimental group to perform better. However, there was no significant difference in the number of attempts on the first four exercises (Hypothesis 2A), indicating the control group was not outperformed due to lack of experience.

The patterns in the application exercises were similar to those in the recall exercises. That is, the code segments required to produce the patterns in both the recall and application exercises required one base case and one recursive case. The differences were in the condition tested to determine the base case and the specific view commands used in the recursive case. There was not a significant difference in the number of attempts on these exercises (Hypothesis 2B), although the difference between the group scores was greater than for the recall exercises.

The major factor in the difference in number of exercise attempts was the transfer exercise (Hypothesis 2C). The code segment required to produce the pattern in this exercise required one base case and two recursive cases. To this point, students in both groups had solved problems with only one base case and one recursive case. Thus, this exercise required a level of understanding of recursion that was beyond the students' previous experience. The fact that the experimental group required fewer attempts than the control group to complete this exercise indicates that the experimental group had a better grasp of the concepts involved in recursion.

One possible reason for the difference between groups on this exercise is that students in the experimental group, by using the Command Lesson, may have better developed their ability to search out the recursive pattern of a problem. When writing a code segment, in this simulation or otherwise, trial and error is often a common approach. However, since the commands are entered individually and the effect of incorrect commands often required students to restart an exercise, most students seemed to carefully consider the effect of each command before execution. The researcher observed this behavior during the lab session. Students were often seen pointing carefully at individual squares on their screens and overheard making comments such as, "Be careful, there is no 'unfill' command." This consideration of each command's contribution to the solution required some understanding of the total pattern. Once this pattern is identified, the base case(s) and recursive case(s) are usually evident. Students in the control group, on the other hand, may not have been as adept at identifying the recursive pattern. Their only experience with these patterns was the recall and application exercises, and their success on those could have been due to the similarity of the exercises.

The fact that there was not a significant difference in the final exam scores of each group is not surprising. The level of success on the final exam is similar to the recall and applications exercises, likely for the same reasons. That is, after working on the set of exercises, solutions to all of the exercises were discussed during the final class period. Thus, solving similar exercises on the final exam only a few days after the final class period would have been more like recall exercises rather than transfer exercises.

The higher scores by the experimental group on the follow-up exam may be due to a better way of thinking about the problems, and therefore a deeper understanding of recursion. Specifically, as indicated by the final exam, both groups were able to write code to create recursive patterns in the 8x8 grid. However, due to

their experiences with the Command Lesson, students in the experimental group were more likely to search for and identify the recursive structure of the pattern before writing their code. In other words, they were using the synthesis model to solve the problems. Students in the control group, on the other hand, may have been relying on more of a syntactic or template model to solve the problems since their original experiences with the patterns were classroom presentation of examples and solutions. Since the follow-up exam was given six weeks after the final exam, students relying on the syntactic or template model would have had a difficult time recalling the appropriate templates. Students using the synthesis model would not have had this problem.

Learning style did not have a significant effect on any of the performance measures, and there was not an interaction between group designation and learning style. One conclusion is that the nature of the simulation makes it an equally valuable learning experience for learners of all types. Others (Cornwell and Manfredo, 1994) argued against the validity of Kolb's learning style measure itself. The first fault found by Cornwell and Manfredo (1994) is that the Learning Style Inventory was originally designed to address management training needs. Therefore, it is best suited to the intended audience of business managers and business school students and is not necessarily an effective measure in other domains. Second, the Learning Style Inventory uses an ipsative scoring format. This format requires students to rank order four learning styles according to preference. Cornwell and Manfredo note that "ipsative scales are not suitable either for psychometric evaluation or for theory testing" (p. 319).

Summary

As the exercises increased in level of difficulty, the difference in the number of attempts required to solve the problems became wider. The number of attempts required for the recall exercises was almost identical for the experimental group (6.88) and the control group (6.92). While not statistically significant ($p < 0.11$), the experimental group needed fewer attempts (4.40) than the control group (5.13) on the application exercises. There was a statistically significant difference ($p < 0.02$) in the number of attempts required by the experimental group (7.00) and the control group (10.50) to complete the transfer exercise. These results indicate that students in the experimental group may have gained a deeper understanding of recursion. This is consistent with the results of Upah and Thomas (1993) and Thomas and Hooper (1991) in that the effects of the treatment are most evident in problems requiring transfer. Students in the experimental group appear to be better able to generalize and transfer their knowledge of the basic patterns to more complex problems.

Scores on the follow-up exam support the findings from the exercise attempts. The experimental group scored significantly higher (10.80) than the control group (8.40) on the follow-up exam. Students who had successfully developed appropriate mental models of recursion would be better able to complete exercises on a follow-up exam given six weeks after the treatment than those relying on inadequate mental models of recursion or memorization.

CONCLUSIONS

This chapter begins with a discussion of the basic questions: "How should students think about recursion?" and "How do students think about recursion?" Next, the strengths and shortcomings of the simulation and lesson plan in helping students think about recursion are presented. Finally, ideas for future research with this simulation and future research on teaching and learning recursion are given.

Basic Questions

The first question of how students should think about recursion was addressed in the literature review. To review, four mental models of recursion were identified: the looping model, the stack model, the syntactic model, and the synthesis model. The looping model is an incorrect model of recursion that students often begin with, but must overcome. The stack model is useful for testing and debugging, but is not necessarily helpful in writing recursive programs. The syntactic model may help students generate a correct solution, but it does not necessarily lead to a deeper understanding of recursion. The synthesis model of recursion is the strongest and most abstract. It requires students to identify the base and recursive cases and most importantly the relationship between the recursive cases and the base case, or how the recursive cases move closer to the base case. The simulations' strengths in developing this model are discussed in the following section. For now, consider the second basic question: How do students think about recursion? To answer this question, first consider how programming is traditionally taught and the perceptions students acquire from that approach.

Traditional approaches to teaching programming begin by deconstructing programs. Program examples are presented and explained by carefully examining the effects each statement has on the internal state of the machine. Students learn that the final or goal state is not reached instantly; rather, the program must progress through many intermediate states. As students learn branching and looping control structures they are taught to trace the program's execution through each branch and each iteration. The result is that in learning to read, write and debug programs, students are taught to view programs as an evolving process where each statement causes something to happen.

Many computer scientists believe abstraction through the use of subroutines is the most powerful tool in programming. Subroutines are usually introduced in a fashion similar to other basic topics, by deconstructing example programs. Students are given an example program with a subroutine and are taught to trace the program as several subroutine calls are made. Students are taught how actual parameter values are bound to formal parameters and how results are communicated from the subroutine to the calling program. The result is that students think of a subroutine call in the same way they think about other programming constructs.

Unfortunately, this is not the best approach to teaching abstraction. While beginning students may become proficient at writing simple subroutines to perform specified tasks, they often struggle to identify the necessary subroutines when solving a larger problem. This is because writing subroutines and using subroutines to solve a larger problem are somewhat contradictory tasks, the first requiring attention to every detail of the interaction and the other requiring high-level abstraction. In other words, teaching subroutines by

deconstructing example programs and tracing in detail the flow of control and parameter passing leads students to view subroutine calls as more work to be done by the program. However, using subroutines to solve larger problems requires viewing subroutine calls not as more work to be done but as a completed task that contributes to the solution of the larger problem.

It is this researcher's opinion that recursion is abstraction at its finest. When writing a recursive subroutine, to be able to include a recursive call to the unfinished subroutine requires a great deal of abstract thinking. Traditional approaches to teaching recursion, as well as this simulation to some extent, reinforce the viewpoint that recursive calls are more work to be done rather than a completed subtask. This is done by spending significant amounts of time and effort teaching students how recursive subroutines execute using the stack model. Thus, as with other programming constructs such as loops, students view recursion, specifically recursive calls, as more work to be done, or an evolving process. With this view it is difficult to see a recursive call as being progress toward the solution.

So, how do students think about recursion? I believe students think about recursion in the same manner they think about programs in general, as an evolving process that progresses through many intermediate states before reaching the goal state. How should students think about recursion? Students should think about recursion as an abstract method of solving a problem. Specifically, when beginning to work on a problem, students search for the base case(s), the recursive case(s) and the recurrence relation. This requires developing the synthesis model of recursion. Then, as students write recursive code, they must view recursive calls as an abstract means of computing part of the final solution.

Simulation

The purpose of this research was to create a simulation that would provide students with experiences useful in developing productive ways of thinking about recursion. In some aspects the simulation was quite successful, in others it was not. The simulation appeared to be successful in helping students develop the synthesis and stack models of recursion. Solving the problems, both in the Command Lesson and the Code Lesson required students to identify the recursive structure of the patterns. This included finding the base case and recursive case as well as identifying the relationship between the two. All of these are fundamental to the synthesis model of recursion. This simulation also provided students with experience using the stack model, although it was not explicitly taught. The windowing environment used to illustrate the execution of the recursive code closely models the stack model of recursion. From this experience, students were exposed to important concepts of the stack model and would likely be able to use it to effectively trace and debug other recursive procedures.

The simulation does however fall short in helping students develop the abstract thinking skills necessary to write recursive code. Part of the reason for this failure may in fact be the reason for the success in teaching the stack model. Demonstrating the execution of the recursive procedures using the windowing environment reinforces the view of recursive calls as more work to be done rather than a completed subtask.

This conflicts with the goal of teaching students to view recursive calls abstractly as part of the solution. At this point there is not a readily apparent way to overcome this paradox.

Future Research

It is important to realize that in research on simulations there is rarely a difference found between groups unless the test exercises require transfer of knowledge (Upah & Thomas, 1993; Thomas & Hooper, 1991). In the set of exercises used in this research, only one problem required transfer. In doing future research, more such exercises should be developed and included. The transfer exercise that was included required students to use one base case and two recursive cases. For future research, perhaps an exercise that required two base cases would be useful.

During this research the Command Lesson and the Code Lesson were treated as separate experiences. While the two lessons are certainly different, they are so closely related that it does not seem appropriate to have either group use one but not the other. Recall that the experimental group used the Command Lesson as their first exposure to recursion and then followed that by using the Code Lesson to solve the set of eight exercises. The control group was given a traditional, lecture-based introduction to recursion and used the Code Lesson only to solve the eight exercises. To better test the effectiveness of both lessons as an introduction to recursion, research needs to be done using both lessons in the experimental group and neither in the control group. Comparisons would then be made on recursive problems from an entirely different domain with both groups receiving the same instruction. This would also ensure that the test exercises required transfer of knowledge from the simulation.

Recursion is an extremely powerful programming technique that is fundamental in a computer science education. However, it is clearly not the appropriate solution for every problem. For example, the patterns used in the simulation could be created with nested loops and conditional statements. If nested loops have been covered in the context of two-dimensional arrays, many students may have a difficult time seeing the need for recursion. Fortunately, there are several areas where recursion is better suited than other techniques. Manipulating list and tree data structures are two examples. Covering recursion separately from such naturally recursive topics severely limits the value students will see in learning it. Thus, while the simulation developed here may be a good method of introducing recursion, more simulations that manipulate recursive data structures are needed. This may have the effect of pushing the topic of recursion into a later programming course where list and tree data structures are studied, but choices as to what material to cover in the introductory course must be made.

Finally, there is a need for research to seek a way to promote abstraction. One approach may be to spend more time with general subroutines before introducing recursion. Note that in order to promote abstraction, students would need to spend more time using defined subroutines to solve larger problems rather than focusing on the details of writing subroutines and tracing their execution. Another approach could be to teach the introductory programming course in a functional language such as Lisp or Scheme rather than an

imperative language such as Pascal or C. Such a change would likely mean less time spent on traditional topics such as iteration. This leads to many questions, including the question in Chapter 1, "Is recursion an advanced topic?" Since abstraction appears to be such a critical part of recursion, the question becomes "Is abstraction an advanced topic?" As noted earlier, many computer scientists feel abstraction is their most powerful tool. Does this mean it is of paramount importance in an introductory course? Would placing emphasis on abstraction allow students to progress from novice to expert more quickly? If abstraction and recursion are studied early in the computer science curriculum, what topics are put off until later? All of these questions need to be answered. The difficulty in teaching abstraction has led many to assume that other topics should come first. Continued development of simulations and other methods of teaching abstraction may change this assumption.

Summary

This research has identified specific areas of recursion that are especially problematic for educators and students. The simulation may be useful in overcoming some of these difficulties, yet it appears to fall short in other areas. Future simulations and future research should attempt to develop in students their skills of abstraction and should measure success by requiring students to write recursive solutions to problems requiring transfer of knowledge.

APPENDIX A. QUIZ

The following quiz on the Towers of Hanoi problem was given during the last class period of the study. The code written in Italics was to be provided by the students. The instructor provided the remaining code. The scores on this quiz were included as part of each students' final exam score.

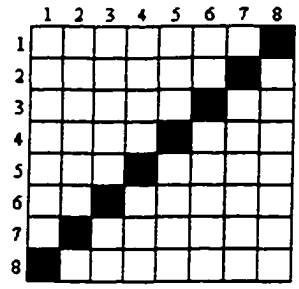
```
// Program: Towers of Hanoi Quiz
#include <iostream.h>

void towers( int numDisks, char startPeg, char destPeg, char sparePeg )
{
  if( numDisks == 1 )
  {
    cout << "Move disk from " << startPeg << " to " << destPeg << endl;
  }
  else
  {
    towers( numDisks - 1, startPeg, sparePeg, destPeg );
    cout << "Move disk from " << startPeg << " to " << destPeg << endl;
    towers( numDisks - 1, sparePeg, destPeg, startPeg );
  }
}

int main()
{
  towers( 7, 'A', 'B', 'C' );
}
```

Grading

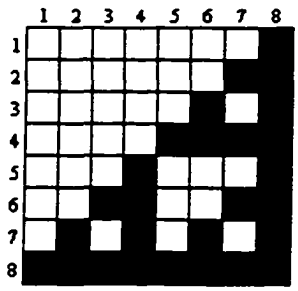
There were five points possible on this question. Students earned one point for each of the following elements of the solution: Use of an if-else statement, correct condition in the if-else statement, correct base case code, correct number of disks parameter in the recursive calls, and correct parameters in recursive calls. Students were not graded on syntax.

APPENDIX B. FINAL EXAM

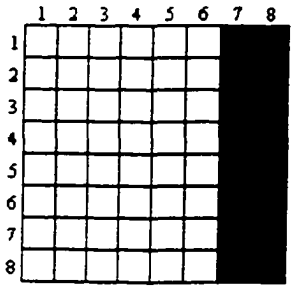
The following is the recursive portion of the final exam. In the first four exercises, students were given recursive patterns and asked to write a recursive code segment that would generate the pattern. Solutions are shown in italics.



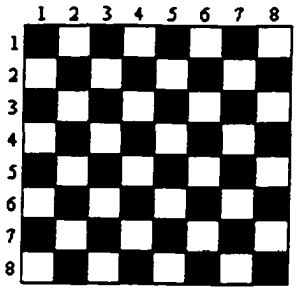
```
if( size == 1 )
  fill
else
  view upper right
  view lower right
endif
return
```



```
if( size == 4 )
  fill
else
  view upper right
  view lower right
endif
return
```



```
if( size == 1 )
  fill
else
  view upper right
  view lower right
  view lower left
endif
return
```



```
if( size == 1 )
  fill
else
  if( size == 4 )
    view upper left
    view lower right
  else
    view upper right
    view upper left
    view lower right
    view lower left
  endif
endif
return
```

The last question of the final exam asked students to determine the output of the following program. The solution is shown in italics.

```
#include <iostream.h>

void towers( int numDisks, char startPeg, char destPeg, char sparePeg )
{
  if( numDisks == 1 )
  {
    cout << "Move disk from " << startPeg << " to " << destPeg << endl;
  }
  else
  {
    towers( numDisks - 1, startPeg, sparePeg, destPeg );
    cout << "Move disk from " << startPeg << " to " << destPeg << endl;
    towers( numDisks - 1, sparePeg, destPeg, startPeg );
  }
}

int main()
{
  towers( 3, 'X', 'Y', 'Z' );
}
```

Solution:

Move disk from X to Y
Move disk from X to Z
Move disk from Y to Z
Move disk from X to Y
Move disk from Z to X
Move disk from Z to Y
Move disk from X to Y

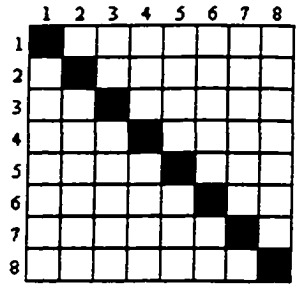
Grading

There were four points possible on each of the first four problems. Students earned one point for each of the following elements of the solution: Use of an if-else statement, correct condition in the if-else statement, correct base case code, and correct recursive case code. Students were not graded on syntax.

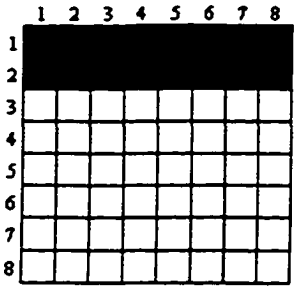
There were four points possible on the last problem. Students earned two points for having the correct number of moves and two points for having the correct peg names in each move.

APPENDIX C. FOLLOW-UP EXAM

The following exam was given to those students enrolled in the second semester course. In the first two exercises, students were given recursive patterns and asked to write a recursive code segment that would generate the pattern. Solutions are shown in italics.



```
if( size == 1 )
  fill
else
  view upper left
  view lower right
endif
return
```



```
if( size == 4 )
  fill
else
  view upper right
  view upper left
endif
return
```

In this problem, students were asked to determine the output of the following program.

```
#include <iostream.h>

int mystery( int n )
{
  if ( n <= 0 )
    return 0;
  else
    return n + mystery( n - 2 );
}

int main()
{
  int result;

  result = mystery( 8 );

  cout << "Result is " << result << "." << endl;
}
```

Soltuion:

Result is 20.

Finally, students were given the following short description of the Fibonacci numbers, along with the partially completed program. Their task was to complete the function definition, shown here in italics.

The following sequence of numbers are called Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
The first and second numbers in the series are defined to be one. All of the remaining numbers in the series are defined to be the sum of the previous two.

Complete the function definition in the program below so that when the program runs and the user enters a number for n, the program will display the n^{th} Fibonacci number. That is, if the user enters 7, the program will print the number 13 because 13 is the 7^{th} number in the Fibonacci sequence.

```
#include <iostream.h>

int fibonacci( int n )
{
  if( n <= 2 )
  { return 1;
  }
  else
  { return fibonacci( n - 1 ) + fibonacci( n - 2 );
  }
}

int main()
{
  int n, result;

  cout << "Enter a number: ";
  cin >> n;

  result = fibonacci( n );

  cout << "The " << n << "th fibonacci number is: " << result;
}
```

Grading

There were four points possible on each of the first two problems. Students earned one point for each of the following elements of the solution: Use of an if-else statement, correct condition in the if-else statement, correct base case code, and correct recursive case code. Students were not graded on syntax.

There were two points possible on the third question. Students earned two points if they correctly calculated the result.

There were five points possible on the last problem. Students earned one point for each of the following elements of the solution: Use of an if-else statement, correct condition in the if-else statement, correct base case code, correct placement of recursive calls, and correct parameters in recursive calls. Students were not graded on syntax.

APPENDIX D. INFORMED CONSENT FORM

To: Computer Science 150 Students, Fall 1997

From: Randall Bower, Assistant Professor, Computer Science Department

Subject: Consent for participation in the following research project.

For many complex programming problems, recursive solutions are often very concise, easily understood, and algorithmically efficient. However, teaching recursive programming has proven to be a difficult task. This research will investigate the effects of a manipulative, graphical model of recursion on the teaching and learning of recursive computer programming. The results of the research could lead to methods of instruction that will help students gain a deeper understanding of recursion than has been previously attained.

This semester as we study recursion the two sections of the course will be taught using slightly different approaches. One section will use a simulation similar to those used earlier in the course. The other section will follow a more traditional approach to teaching recursion that uses classroom lectures and discussion of examples. Following this initial instruction, you will be asked to complete several recursive exercises in lab. As you work on these exercises, your methods of solving the problems and success rate will be recorded. You will also be given some paper and pencil exercises that will be graded and used in the research. Participants will likely benefit by thinking about and gaining insights into recursion as well as obtaining experience writing recursive programs.

Any evaluation or grading of your work will be adjusted to reflect the group to which you were assigned. There are no known risks involved in this study and there will be no time commitment above and beyond the regular requirements of the course.

All information obtained will be held confidential to the researchers (principal investigator and major professor) and the research participants. This information will be disclosed only with your permission. Names used in the resulting Ph.D. dissertation or in any publications or presentations will maintain confidentiality through the use of pseudonyms. All names will be removed from all data gathering instruments by 12/15/97.

Consenting to the use of your data for this study is completely voluntary. You may withdraw your consent at any time without prejudice. If you choose not to give your consent, you will be allowed to select the form of instruction you receive. If you have questions or concerns about this research at any time, you are free to contact me and I will do my best to address your concerns.

If you are willing to help try to find a better way to teach and learn recursion, please sign your name on the line below.

Thank you very much.
Randall Bower, Assistant Professor, Computer Science Department

Signature_____ Date_____

REFERENCES

Anderson, J., Pirolli, O., and Farrell, R. (1988). Learning to Program Recursive Functions. In M. Chi, R. Glaser, & M. Farr (Eds.), The Nature of Expertise (pp. 153-183). Hillsdale, NJ: Erlbaum Associates

Anzai, Y. and Uesato, Y. (1982). Learning recursive procedures by middleschool children. Proceedings of the Fourth Annual Conference of the Cognitive Science Society (pp. 100-102).

Astrachan, O. (1994). Self-reference is an illustrative essential. ACM SIGCSE, 26(1), 238-242.

Bhuiyan, S., Greer, J., and McCalla, G. (1989). Mental models of recursion and their use in the SCENT programming advisor. In S. Ramani, R. Chandrasekar, and K. Anjaneyulu (Eds.) Knowledge-based Computer Systems (pp. 135-144). New Delhi, India: Narosa Publishing House.

Bhuiyan, S., Greer, J., and McCalla, G. (1991). Characterizing, rationalizing, and reifying mental models of recursion. Proceedings of the Thirteenth Cognitive Science Society Conference (pp. 120-126).

Bhuiyan, S., Greer, J., and McCalla, G. (1992). Learning recursion through the use of a mental model-based programming environment. Proceedings of the Second International Conference on Intelligent Tutoring Systems (pp. 50-57).

Bhuiyan, S., Greer, J., and McCalla, G. (1994). Supporting the learning of recursive problem solving. Interactive Learning Environments, 4(2), 115-139.

Billstein, R. and Moore, M. (1983). Recursion, recursion. The Computing Teacher, 11(5), 46-47.

Brant, G., Hooper, E., and Sugrue, B. (1991). Which comes first, the simulation or the lecture? Journal of Educational Computing Research, 7(4), 469-481.

Cornwell, J. and Manfredo, P. (1994). Kolb's learning style theory revisited. Educational and Psychological Measurement, 54(2), 317-327.

Dicheva, D. and Close, J. (1996). Mental models of recursion. Journal of Educational Computing Research, 14(1), 1-23.

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

Gibbons, P. (1995). A cognitive processing account of individual differences in novice LOGO programmers' conceptualization and use of recursion. Journal of Educational Computing Research, 13(3), 211-226.

Hooper, E. (1986). Using programming protocols to investigate the effects of manipulative computer models on student learning. Diss. Abstr. Int., 47, 3009A. (University Mocrofilms No. DA8627118.)

Hooper, E. and Thomas, R. (1990). Investigating the effects of a manipulative model of computer memory operations on the learning of programming. Journal of Research on Computing in Education, 22(4), 442-456.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway (Ed.), Studying the Novice Programmer (pp. 209-228). Hillsdale, NJ: Erlbaum Associates.

Kessler, C. and Anderson, J. (1986). Learning flow of control: Recursive and iterative procedures. Human-Computer Interaction, 1, 135-166.

Kolb, D. (1985). Learning-Style Inventory. McBer & Company, Boston, MA.

Kurland, D. and Pea, R. (1985). Children's mental models of recursive LOGO procedures. Journal of Educational Computing Research, 1(2), 235-243.

Lough, T. (1983). A cure for recursion. The Computing Teacher, 11(5), 34-37.

McCracken, D. (1987). Ruminations on computer science curricula. Communications of the ACM, 30(1), 3-5.

Moore, M. (1983). A recursion excursion with a surprising discovery. The Computing Teacher, 11(5), 49-52.

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

Riordon, T. (1983). Helping students with recursion: Teaching strategies. The Computing Teacher, 11(5), 38-42.

Riordon, T. (1984). Helping students with recursion: Teaching strategies part II: Moving to the computer. The Computing Teacher, 11(6), 59-63.

Riordon, T. (1984). Helping students with recursion: Teaching strategies part III: Teaching students about embedded recursion. The Computing Teacher, 11(7), 64-69.

Roberts, E. (1986). Thinking Recursively. New York, NY: Wiley & Sons, Inc.

Thomas, R. and Boysen, P. (1984). A taxonomy for the instructional use of computers. Monitor, June 1984, 15-26.

Thomas, R. and Hooper, E. (1991). Simulations: An opportunity we are missing. Journal of Research on Computing in Education, 23(4), 497-513.

Troy, M. and Early, G. (1992). Unraveling recursion part I. The Computing Teacher, 19(6), 25-28.

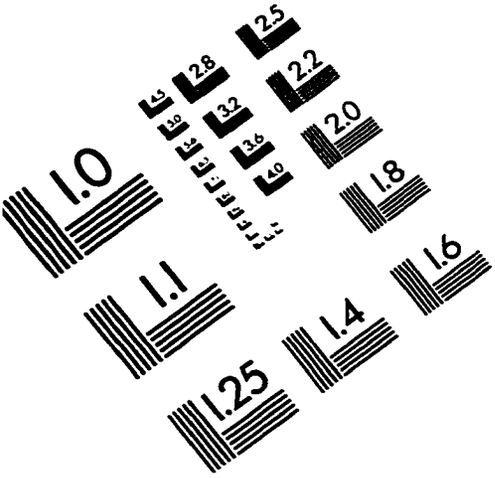
Troy, M. and Early, G. (1992). Unraveling recursion part II. The Computing Teacher, 19(7), 21-25.

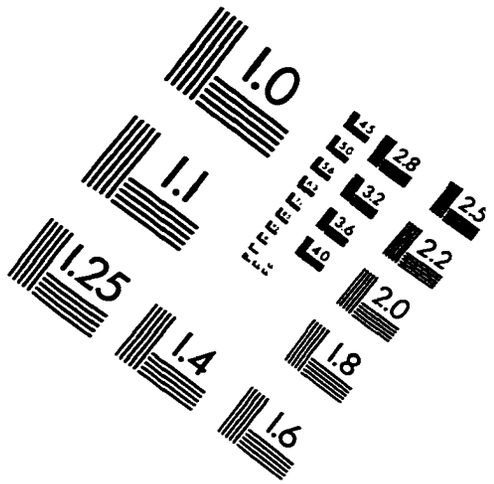
Upah, S. & Thomas, R. (1993). An investigation of manipulative models on the learning of programming loops. Journal of Educational Computing Research, 9(3), 397-412.

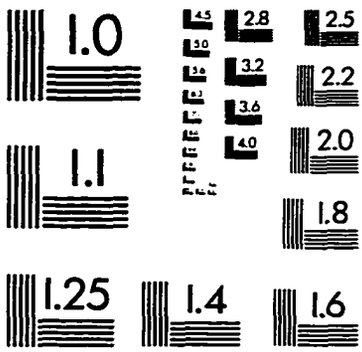
Widenbeck, S. (1988). Learning recursion as a concept and as a programming technique. ACM SIGCSE, 20(1), 275-288.

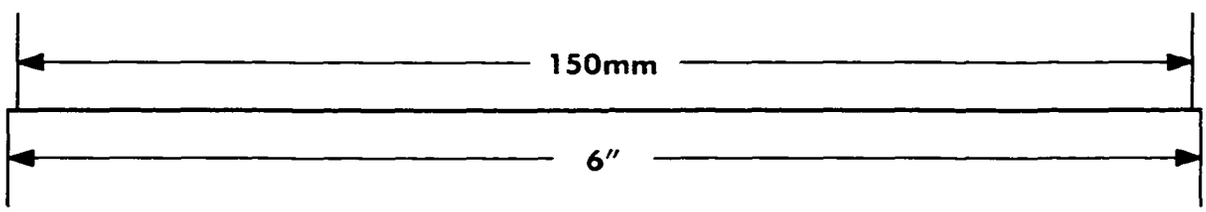
Wilcocks, D. and Sanders, I. (1994). Animating recursion as an aid to instruction. Computers in Education, 23(3), 221-226.

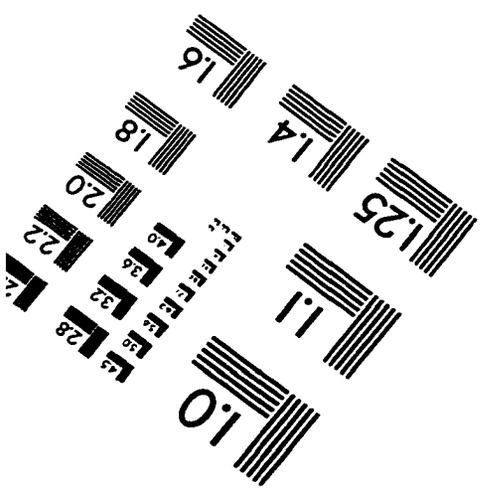
IMAGE EVALUATION
TEST TARGET (QA-3)

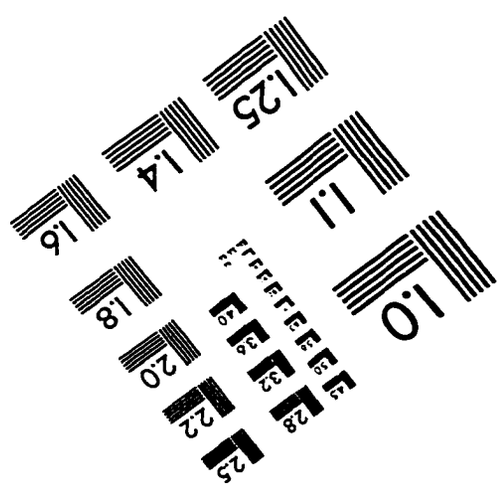












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989