AN EVALUATION OF INSTRUCTIONAL METHODS FOR TEACHING RECURSION

Tia Shelley

In most fields of study, there is some topic that students historically struggle with. In some, this subject may force students to remain at that level until understanding dawns, or more ominously, until they choose to switch fields. Other topics challenge students to a different extent, in that the student may manage to squeak by for a semester without ever really grasping the topic. There are some ways in which this second group of topics is more dangerous, and it is more interesting in regards to how those students manage to continue within the discipline after that class. Will they side-step that topic through their entire undergraduate career and manage to graduate without some building block of knowledge that is considered fundamental by their instructors?

In computer science, the building block most often left out of this metaphorical tower of learning is recursion. Students typically struggle with this topic, forming models that incorrectly represent how recursion works, while applying the primary rules correctly. Painful as it is to consider, a small number of computer science majors may graduate without real comprehension of how recursion works, because of such "working" models that are formed early on and are difficult to replace. It is the intent of this research to explore recursion and how it is currently being taught in the American university system.

This work will be of value primarily to those students seeking a comprehensive explanation of recursion and to Computer Science educators. However, those interested in education generally may find this study interesting as an exploration of instructional methods. Many disciplines have subjects that students repeatedly struggle with, and educators in those fields may also benefit from an exploration of the teaching of complex material in the field of Computer Science.

The research presented here will synthesize a selection of works that relate to recursion, generally studies performed by computer science educators on methods of teaching recursion, as well as referencing texts that are used in introductory courses. If there is some method of teaching recursion that results in a higher success rate of students who continue on within a degree program in computer science, clearly that method needs to be supported. Additionally, methods that result in semi-viable or non-viable models should be shown to be ineffective. If recursion is not being taught in a manner that results in viable models, then computer science as a field will suffer from a lack of powerful alternative problem-solving methods. The researcher has found that it is quite possible to complete an undergraduate education in Computer Science without a fully functional model of recursion. This study aims to address this issue.

The specific purpose of the study is to describe fully the recursive concept and to discuss methods that instructors use to teach recursion. Correction of non-viable models will not be offered here; instead, this paper presents some discussion of contemporary forms of teaching recursion and the advantages or disadvantages those current methods may provide to students. Though certainly of interest to the topic, questions of a technical nature such as the recursion-first versus iterative-first debate have been excluded as somewhat divergent from the primary focus of this paper. Similarly related to Computer Science instruction but also not

included in this study is a discussion of which language or types of language are of maximum benefit to aid student comprehension.

Terminology

Mental Model (also simply Model) – This term is typically used to describe or define the perception of a process by a particular individual (Norman)/

Recursion - A more complete discussion of recursion will follow, but the simplest definition is that it is a process that is capable of calling additional instances of itself to perform other portions of some task.

Active Flow – This process is the flow of control as a recursive process calls the additional instances of itself.

Passive Flow – This process returns control to the process that initially created that instance once the task has been completed.

Methods - In computer science, "methods" are portions of code that can be called with different parameters that can perform an action and output or return a value. A method is non-functioning until it is instantiated. An instance of a method implies that it has been given some parameters and will perform some actions on those parameters and possibly output a certain value based on them.

Pseudocode – This planning step is used in computer science before the coding process begins. As writers use outlines to plan their papers, programmers use pseudocode to plan programs. Just as an outline does not have complete sentences, pseudocode is not computer-readable.

Methods

The research on recursion has been primarily conducted by university instructors, and examination of student comprehension is the primary focus of their research. Most of the sources consulted were published in the last decade, though the information referenced in the brief discussion of recursion below, designed for those unfamiliar with computer science, spans an additional three decades. This particular selection of sources was designed for an inductive approach, aiming to find a unifying theme in recursion instruction. Thus the analysis performed on the texts was largely comparative, focusing on how effectively the concept of recursion was explained within the text.

Presentation of Findings

I willl begin with a summary of how recursion works, as described both in the textbooks and in the articles discussing methods of teaching recursion. I will also discuss some additional findings on recursion instruction.

Recursion is a process capable of calling additional instances of itself to complete a task. To explain this concept, an metaphorical example is possibly the best way of conveying a recursive-type task. Rather than using a mathematical example, this paper will attempt to use some of the lessons learned from other studies to explain recursion through more natural models, and then link the subject to computer science more specifically.

When a task is sufficiently large, it can often be difficult for one individual to complete the task. Rather than attempting to complete the entire task, the individual may choose to delegate parts of the task to other people. When the chunks of the task have been handed off, the delegating individual must then wait until those chunks of the task have been completed. The people who received the smaller tasks might still find them too large, and thus hand off portions of their task to more people, and have to wait for those sub-tasks to be returned.

At the very bottom of this chain of command, the individuals have very small problems to solve. These can no longer be

broken down any further, so these individuals are at what will be called the base case. They simply do their part of the task, and hand their completed sub-task back up a level to the one who asked them to complete the task. That person takes all of the data handed to them and performs whatever action is needed to complete whatever task was given them and hands that back up. This continues until the original delegating individual has performed their final action on the completed tasks returned to them.

This passing of tasks to other methods and the implied dependencies generated by it is a schematic introduction to recursive concepts. Figure 1 illustrates what is described above in a simplistic fashion. At this point, it is important to flesh out these concepts and apply them to computer science. This will be accomplished through exploring the pseudocode of a classic recursive problem and then following a trace of that problem.

Quite possibly the classic recursion problem is that of "n-factorial," written henceforth as n!. Defining the problem in a forward thinking format provides: n! = 1 * 2 * 3 *...* (n-1) *n. Defining the solution to this mathematical function in a recursive format provides: n! = n * (n-1)!. There is one additional part of the definition: 0! =1. This base case provides an end situation so that calculations do not become infinite (and thus meaningless here). Some pseudocode follows that could be used as a planning tool for an n! problem:

factorial (n)

if n = 0, then 1 is returned.

else, n * factorial (n-1) is returned.

This pseudocode closely resembles the definition provided previously. To gain additional understanding of the principles that are being shown, a trace of the "program" being called with an n value of four will be shown. The complete article includes a visual trace of the process described below.

In the first step of the process, the method recognizes that the value of 'n' is not equal to zero. Therefore, it must call another method to do the next step. The control passes from the first factorial method to the second, calling on the second to do additional work. At this stage, n still does not equal 0, and thus, factorial is called additional times. When the base case is arrived at (which is to say, when n =0), the active control flow is halted. Control no longer passes "downward." Now it begins to head back "up."

From this point, passive flow is in effect. Each method returns the value it evaluated "up" as it receives the appropriate information from the method it called. Note that each instance returns the value to the instance that called it initially.

Tracing the process in this way provides a good deal of information about how recursion works in programming. First, that when a call is made, additional instances are created. This implies that these instances will need space to exist. Secondly, it shows that the instances do not stop existing while waiting for the "lower" methods to perform their calculations. The instances all coexist, which increases memory use during the recursive process.

After this introductory explanation of recursion, it is now appropriate to discuss recursion instruction as it currently exists. There seem to be several schools of thought as to how recursion is best taught, which incorporate at least three similar models of presentation arranged in different orders.

The first model of presenting recursion is through metaphor (Edginton, 2007, Wirth, 2008). The strength in using metaphors is primarily in their accessibility to students who may not have been exposed to recursion in mathematics or linguistics or who have only been introduced to the topic tangentially. Critical to whether or not metaphor use aids

the student to obtain a correct mental model of recursion is how closely the metaphor can be mapped on to the process. Metaphors that do not demonstrate all of the aspects of recursion may cause incomplete models to be formed.

A second model of presenting recursion is through classical mathematical problems. This method is seen most typically in textbooks (Liang, 2005, Carrano & Prichard, 2006, Shiflet & Nagin, 2003). Towers of Hanoi, n!, Fibonacci numbers, reversing a string...all of these problems have been frequently used to introduce students to problems for which recursion can be a simpler solution than iteration. Recursion taught through induction principles builds on a strong mathematical background. However, students with weak mathematical backgrounds may struggle with both the problems and the problem-solving concepts underlying them.

Thirdly, there is the visual/kinesthetic method of presenting recursion (Carrano & Prichard, 2006, Shiflet & Nagin, 2003). By tracing recursive programs during instruction as in the above diagrams or by having students act out recursion in skits, instructors can better assist learners who benefit from other modes of presentation in order to grasp abstract topics in general. Once again though, there is the risk of misunderstanding if the dialogue that guides or accompanies the traces or the acted-out skits is not clear.

It is rare that any one of these models is used without the accompaniment of at least one of the others. The strength of the multiple-model lesson plan is in the number of students to whom it can communicate clearly: if multiple methods are used and time is spent on the presentation, it is more likely that a larger number of students will be able to form correct mental models of the topic.

Conclusion

The exploration of recursion and how it is being taught is part of an ongoing discussion. Education is a complex process in any subject, and doubtless many more variations on traditional instructional methods will continue to arise as today's students become the teachers of the future.

Works Consulted

Carrano, F., & Prichard, J. (2006). Data Abstraction and Problem Solving with Java. Boston: Pearson/Addison Wesley.

Edgington, J. (2007). Teaching and viewing recursion as delegation. J. Comput. Small Coll. 23, 1 (Oct. 2007), 241-246.

Fox, J., (1982). Software and Its Development. Englewood Cliffs: Prentice-Hall.

Ginat, D. (2004). Do senior CS students capitalize on recursion?. In Proceedings of the 9th Annual SIGCSE Conference on innovation and Technology in Computer Science Education (Leeds, United Kingdom, June 28 - 30, 2004). ITiCSE '04. ACM, New York, NY, 82-86

Götschi, T., Sanders, I., and Galpin, V. (2003). Mental models of recursion. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (Reno, Navada, USA, February 19 - 23, 2003). SIGCSE '03. ACM, New York, NY, 346-350.

Kay, J. S. (2000). Using the force: how star wars can help you teach recursion. In Proceedings of the Fifth Annual CCSC Northeastern Conference on the Journal of Computing in Small Colleges (Ramapo College of New Jersey, Mahwah, New Jersey, United States). J. G. Meinke, Ed. Consortium for Computing Sciences in Colleges. Consortium for Computing Sciences in Colleges, 274-284.

Liang, Y., (2005). Introduction to Java Programming. Upper Saddle River: Pearson Prentice Hall.

Norman, D., (1988). The Psychology of Everyday Things. New York: Basic Books.

Rubio-Sánchez, M., Urquiza-Fuentes, J., and Pareja-Flores, C. (2008). A gentle introduction to mutual recursion. In Proceedings of the 13th Annual Conference on innovation and Technology in Computer Science Education (Madrid, Spain, June 30 - July 02, 2008). ITiCSE '08. ACM, New York, NY, 235-239.

Sanders, I., Galpin, V., and Götschi, T. (2006). Mental models of recursion revisited. Proceedings of the 11th Annual SIGCSE Conference on innovation and Technology in Computer Science Education (Bologna, Italy, June 26-28, 2006). ITICSE '06. ACM, New York, 138-142.

Shiflet, A., & Nagin, P. (2003). Problem Solving in C++. Cambridge: Course Technology.

Weiland, R., & Bauer, C. (1983). The Programmer's Craft. Reston: Reston Pub. Co.

Wirth, M. (2008). Introducing recursion by parking cars. SIGCSE Bull. 40, 4 (Nov. 2008), 52-55.