Teaching Recursion in a Procedural Environment - How much should we emphasize the Computing Model?

David Ginat
Science Teaching Department
Weizmann Institute of Science
Rehovot, Israel
ntginat@weizmann.weizmann.ac.il

Eyal Shifroni
Tel-Hai College and
Center of Educational Technology
16 Klausner St. Tel-Aviv, Israel 61394
eyal_s@cet.ac.il

1. Abstract

Recursion is a powerful and essential computational problem solving tool, but the concept of recursion is difficult to comprehend. Students that master the conventional programming construct of iteration in procedural programming environments, find it hard to utilize recursion.

This study started as a test of CS College students' utilization of recursion. It was conducted after they have completed CS1, where they studied recursion with the C programming language. The test revealed that students adhere to the iterative pattern of "forward accumulation", due to their confidence with the iteration construct, but lack of trust of the recursion mechanism. These results motivated us to get more insight into the nature of recursion difficulties and ways to overcome them.

In this paper we describe the difficulties we observed, and present a declarative, abstract, approach that contributed to overcome them. We question the emphasis that should be put on the basic computing model when presenting recursion, and argue for emphasis on the declarative approach for teaching recursion formulation in a procedural programming environment.

1.1 Keywords

Recursive formulation, problem decomposition.

2. Introduction

Recursion is an essential and unique tool for computational problem solving - it encapsulates decomposition of a problem into subproblems of the same kind.

Although such decomposition is logically sound, it is not easily comprehended - the problem solver has to carefully specify decomposition to subproblems and composition of the subproblem solutions.

In the CS1 course with procedural programming, recursion is often taught at a fairly late stage, in a rather concrete level, where an attempt is often made to base comprehension on the students' conception of the computer's *basic computing model.* The progress along the course includes an evolving picture of the basic computer mechanism - the role and scope of variables, assignment, conditionals, and branching. As the picture evolves, the students learn to trace program execution, and develop a conception of the basic computing model. When recursion is introduced, teachers tend to capitalize on that conception and establish comprehension via understanding of the process of recursion execution. This is often done at the expense of de-emphasizing the declarative, abstract, formulation of recursion.

In this paper we present a study that questions this trend. Various studies in recent years concentrated in tracing and animating recursion execution, as we briefly describe below. But, how important is it to emphasize the recursion execution mechanism, in order to enhance the ability to formulate recursive programs?

We observed that emphasis on the concrete level (mechanism) yielded only limited understanding of the computing model with respect to recursion, and caused confusion. On the other hand, emphasis on the declarative, abstract, level considerably improved recursive program formulation.

Considerable research was conducted to study the cognitive difficulties of recursion comprehension. Kahny and Eisenstadt [2] examined novices' judgments of given recursive programs and concluded that they developed one of several mental models of recursion, which they named "copies", "loop", "odd", "null", and "syntactic magic". All of these models except for the "copies model" are regarded as incorrect models of recursion. Kurland and Pea [5] observed programmers that viewed recursion as iteration.

Anazi and Uesato [1], and Kessler and Anderson [4] studied transfer abilities from iteration to recursion and vice versa. They concluded that it is more sensible, pedagogically, to base understanding of recursion on iteration (than iteration on recursion).

But should recursion be taught along the same lines by which iteration is taught - with the picture of the computing model in mind? Iteration is rather simple to trace. It can be viewed as "an accumulation process", where a "pass" starts after its preceding "pass" ends. Recursion, however, is much harder to trace. It is a process where an instantiation starts, and ends, before its preceding instantiation ends.

Segal [6] identified the misconception of "base-case as stopping condition (of execution)" and argued for the importance of recursive function evaluation. Wilcocks and Sanders [7], and Kann et al [3], showed that animation which illustrate the "copies model" of recursion can enhance comprehension and recursive function evaluation. But, to a questionnaire given by Wilcocks and Sanders, most students indicated that the animator did not assist in "being able to develop recursive algorithms to solve problems".

We believe that the key emphasis in enhancing recursion formulation should be at the abstract level of problem decomposition. That is, divide-and-conquer at "the problem level", irrespective of the machine implementation.

In the next section we describe our study of two attempts with recursive formulation. The study started as a test of CS College students' utilization of recursion at the beginning of the CS2 course. The students have completed CS1 where they studied recursion with the C programming language. The test revealed a surprising number of errors in recursive formulation for a simple task - multiplication by consecutive additions. These errors stemmed from the limited understanding of the concrete level with respect to recursion.

We divided the students into two groups. The first group continued studying with emphasis on the concrete level, and the second - with emphasis on the declarative, abstract level. Six weeks later we conducted a second test, in which the task was to compute the various ways to climb a ladder. The test results showed considerable improvement among the second ("declarative") group students, but continued difficulty among the first group students.

3. Study Description

The subjects of this study were 42 computer science college students beginning their second year. The study was conducted in the CS2 course, which includes elaboration of the concept of recursion.

The study was conducted in two phases. Phase A was the test given at the beginning of the course. This test reflected the students' difficulties in recursive formulation. Phase B took place during the following six weeks, and was aimed at examining the success of the declarative approach. In the beginning of phase B we divided the students into group-1, with 22 students, and group-2, with 20 students.

Group-1 was used as a control group - its students continued studying recursion with emphasis on the mechanism of the recursive process. Group-2 was taught by one of us, with emphasis on the declarative level.

In what follows, we present the two phases of the study in more details, and describe our findings.

3.1 Phase A: First Attempt with Recursion

The following twofold task was given to the students in the first class of the CS2 course:

Task1: Multiplication

*1a: Write an **iterative** C function* int mult(int m, int n) *for multiplying two natural numbers by consecutive additions. The function should return the product* m*n.

*1b: Repeat task 1a, but with **recursion** instead of iteration.*

The multiplication task was chosen because it is a simple task with a straightforward iterative and recursive solutions. It is not one of the classical examples (like factorial) that are used to introduce recursion. As such, it could serve to show how the students can apply the concept of recursion in a simple novel task. The iterative part was added to check the students' mastery of basic programming constructs (like assignment, alternation and iteration).

A proper solution for task 1b is:

```
int mult(int m, int n)
{
  if (n == 0)
     return 0;
  else return m + mult(m, n-1);
}
```

This solution includes the *decomposition* of mult(m, n) to the simpler instance mult(m, n-1), and the *incorporation* of the solution of the simpler instance via the addition operation. The *base case* is defined for "n equals 0".

3.1.1 Results

The surprising results were that although most of the students provided proper solutions for part 1a, the majority (32 of the 42) provided erroneous solutions for part 1b (the recursive part). Examples 1-3 below, illustrate difficulties that seem to stem from attempts to formulate recursion based on limited understanding of the computing model with respect to recursion.

In example 1 the recursive call is embedded in a while loop. The loop has no effect here because the only statement it contains is the return statement. This student was probably unsure about the effect of the recursion, and hence preferred to use the familiar iterative construct:

Example 1
```
int mult(int m, int n)
{
 while (n != 0)
     return (m + mult(m, n-1));
}
```

A considerable amount of the solutions show the tendency to use the procedural programming patterns of counting and accumulation:

Example 2
```
int mult(int m, int n)   /* Iterative solution */
{
 int sum=0;
 while (--m > 0)
     sum += n;
 return(sum);
}

int mult(int m, int n)   /* Recursive solution */
{
 int sum=0;
 if (m == 0)
     return(sum);
 else mult(sum+n, --m);
}
```

Note how the counting (--m) and the accumulation (sum +=n) operations are transferred to the recursive solution. The recursive call is not really a decomposition of the problem, but rather a different way to express counting and accumulation.

In the recursive solution above, the accumulation will not work because sum is a local variable; this can be handled using static or global variables:

Example 3
```
int count=0, sum=0;  (global)
int mult(int m, int n)
{
 if (count < n) {
     sum = m + sum;
     count++;
     mult(m, n);
 }
 else return (sum);
}
```

The function defined in example 3 produces the correct result; technically it is a recursive function, but there is no recursive decomposition at all, and the value returned from the recursive call is not used. In fact, this recursive call is used here merely as a goto statement, which handles the repetition.

In examples 2 and 3, the accumulation is done during the nested recursive invocations (forward accumulation), and not during the return (reverse accumulation). This suggests that the students view the base case as a stopping condition of iteration ([6]).

The solutions to task 1a demonstrate that the students were familiar with the computing model, to the extent of the basic programming constructs. But, they could not apply it with respect to recursion formulation. Unfortunately, many of them adhered to the iterative pattern of forward accumulation (often with global variables) in solving task 1b.

3.2 Phase B: Second Attempt with Recursion

In the twice-a-week lab hours of CS2, the class was divided into two groups: group-1, which continued studying recursion with emphasize on the computing model, and group-2, which studied recursion with emphasize on declarative formulation. The students of group-2 were encouraged to explicitly formulate a solution, using their own words, *before* attempting to program it. They were instructed to use divide-and-conquer according to the following guidelines:

1. Define what an instance of the problem is, and how can it be decomposed into simpler instances of the problem.
2. Suppose you already have a solution(s) for an instance(s), define how to incorporate it in the solution of the more general instance.
3. Verify that consecutive simplifications of the general instance yield a basic instance(s) that can be solved directly.

In the sixth week of the semester, we assigned the students the following task, which we refer to here as Task2:

Task2: Ways to climb a ladder

Compute how many different ways are possible to climb an N stage ladder, if one can climb 1 or 2 stages in each step. For example, a 3-stage ladder can be climbed using the three following ways: 1-1-1, 1-2 and 2-1. Write a recursive C function int ways(int n) *that performs this computation.*

The ladder problem is not trivially solved with iteration. But it has an inherently divide-and-conquer solution, which yields a natural recursive formulation.

The reasoning required is as follows: in order to climb an N stage ladder it is possible to climb an N-1 stage ladder and then the remaining stage; or to climb an N-2 stage ladder and then the remaining two stages in one step. A declarative recursive formulation would be:

$$\text{ways}(N)=\begin{cases} N & \text{if } N=1 \text{ or } N=2 \\ \text{ways}(N-1)+\text{ways}(N-2) & \text{if } N>2 \end{cases}$$

3.2.1 Results

The most notable difference between the two groups was the number of students that did not supply an answer: in group-1 - 11 students out of 22; in group-2 - only 5 out of 20.

Out of the 11 students from group-1 that provided answers, only 4 provided proper recursive decomposition. The rest showed difficulties similar to those observed in the solution to Task 1b. We present below some typical examples.

Example 4 below is similar to Example 1 in which the recursive call is embedded in a loop:

Example 4
```
int ways(int n)
{
 int x=2;
 if (n == 1) return 1;
 if (n == 2) return 2;
 else while (++x < n)
               return ways(x-1) + ways(x - 2);
}
```

Clearly, this student did not trust the recursion to do the job and preferred to use the more familiar iterative construct.

Example 5 is similar to examples 2 and 3, because the value returned from the recursive call is not incorporated in the result and the returned value is accumulated in a static variable:

Example 5
```
int ways(int n)
{
     if (n==0) {          // end of ladder
       static int w=0;
       ++w;               // one more way
       return w;
     }
     if (n-2 >= 0)        // more than one stage
           ways(n-2);
     ways(n-1);
}
```

The function defined above returns the correct value, because the statement ++w is executed at each "leaf" instantiation of the recursion. The author of this code probably visualizes recursion as a process that is capable of triggering new instantiations of itself (Kahney's "copies model"), but he does not demonstrate ability to formulate recursive decomposition - a limitation that will probably hinder his ability to solve more complex problems.

Out of the 15 students from group-2 that provided answers to the problem, 8 provided proper solutions. The solutions of the remaining students included minor errors and inaccuracies. A typical example is the following:

Example 6
```
int ways(int n)
{
 if (n == 0) return 0;
 if (n == 1) return 1;
 else return ways(n-1) + ways(n-2);
}
```

In this example the value returned for the base case "n equals 0" is incorrect.

In spite of the minor error, the above typical example reveals improved ability of recursion formulation. This shows that the declarative approach contributed in overcoming difficulties that were encountered in phase A.

4. Discussion

We presented a study of students' difficulties in recursion formulation that stemmed from limited comprehension of the basic computing model with respect to recursion. We also showed that teaching recursion with an emphasis on the declarative, abstract, level of recursion considerably improved the student's ability.

The difficulties revealed in our study demonstrate that students adhere to the iterative pattern of "forward accumulation", due to their confidence with the iteration construct, but lack of trust and full understanding of the recursion mechanism. This phenomenon is enhanced particularly in a procedural programming environment, as with the C language in our study, since teachers of this environment tend to emphasize the basic computing model in teaching the various programming constructs.

Various studies in recent years concentrated in enhancing recursion evaluation ability, by deepening student understanding of recursion tracing. Although significant for understanding the recursion execution process, this emphasis seems to contribute rather little to recursion formulation.

We demonstrated that emphasis on the declarative, abstract, level significantly improved recursion formulation ability. In our study, we outlined the divide-and-conquer guidelines that we offered the students learning with the declarative approach. The students who followed these guidelines managed to avoid the difficulties they encountered earlier, when they adhered to the concrete level of the recursion mechanism. Thus, although we did not address their misconceptions of the computing model with respect to recursion, we managed to bypass difficulties and to considerably improve the students' ability of problem solving with recursion.

Although experts are hypothesized to posses the mental "copies model" of recursion ([2]), there is no evidence that this is the *only* model they posses with respect to recursion. Experts may possess other mental models of recursions and apply each one according to the task at hand. For example, it is possible that experts apply the "copies model" when debugging recursive programs, but apply a different model when formulating a recursive solution.

It is generally agreed that problem solving requires abstraction. While abstraction is rather natural in functional and logic programming, it is less inherent in procedural programming, since procedural languages are conceptually closer to the basic computing model. We believe that in the attempt to alleviate novices to the level of experts, teachers of recursion in procedural programming should firstly emphasize the declarative, abstract, level of divide-and-conquer, and beware of the tendency to strongly relate to the basic computing model in teaching recursion formulation.

5. References

[1] Anazi, Y. and Uesato, Y. "Learning Recursive Procedures by Middle-School Children", *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pp. 100-102, 1982.

[2] Kahney, H. and Eisenstadt, M., "Programmers' mental Models of their Programming Tasks: The Interaction of Real World Knowledge and Programming Knowledge", *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pp. 143-145, 1982.

[3] Kann, C., Lindeman ,R. and Heller, R., "Integrating Algorithm Animation into a Learning Environment", *Computers Educ.* Vol 28, No 4, pp 223-228, 1997.

[4] Kessler, A. and Anderson, J., "Learning Flow of Control: Recursive and Iterative Procedures", *Human-Computer Interaction,* 2, pp. 135-166, 1986.

[5] Kurland, D. M. and Pea, R. D., "Children's mental Models of Recursive Logo Programs", *Proceedings of the Fifth Annual Conference of the Cognitive Science Society*, pp. 1-5, 1983.

[6] Segal, J. "Empirical Studies of Functional Programming Learners Evaluating Recursive Functions", *Instructional Science* 22:385-411, 1995.

[7] Wilcocks, D. and Sanders, I. "Animating Recursion as an Aid to Instruction", *Computers Educ.* Vol 23, No 3, pp. 221-226, 1994.