Appears in the Journal of Computing in Small Colleges,
Volume 14, Numer 4, May 1999, pp 86-101

TEACHING RECURSION BEFORE LOOPS IN CS1

Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst
Computer Science Department
Wellesley College
Wellesley MA, 02481
{fturbak,croyden,jstephan,jherbst}@wellesley.edu
(781) 283-{3049, 2743, 3152, 3162}

ABSTRACT

Traditionally, loops are a central topic in a CS1 course but recursion is viewed as an advanced topic that is either taught near the end of the course or not taught at all. Taking a cue from the function-oriented programming community, we argue that there are strong pedagogical reasons for teaching recursion *before* loops in a CS1 course, regardless of what programming paradigm is taught. In our approach, recursion is presented as an instance of the classic "divide, conquer, and glue" problem solving strategy. Iteration is then presented as a particular pattern of recursion. Finally, loop constructs are presented as concise idioms for iterative patterns. We describe our positive experience in adopting this approach in a CS1 course and highlight the aspects of the course that we think contribute to its success.

1. INTRODUCTION

Discussions of CS1 pedagogy are enlivened by perennial debates on the choice of programming language and the choice and order of topics. The area of topic order is marked by various early/late debates: Should procedures be taught early or late? Should objects be taught early or late (if at all)? In this context, it is surprising that there has been relatively little debate on where (or whether) recursion belongs in CS1. We wish to spark a new debate on whether recursion should be taught early or late in CS1. Our position is that recursion should be taught early and should be taught before loops.

This idea is unusual, if not heretical, but it is not without precedent. Many function-oriented languages (e.g., Scheme, ML, and Haskell) do not have imperative looping constructs. Instead all iterations are expressed via *tail recursion*, a particular form of recursion (see Section 3.1). Textbooks teaching such languages (e.g., [AbSu95, Bir98, Pau96]) necessarily present recursion before iteration.

(Note: In our discussion, it is necessary to distinguish iterative computational processes from the syntactic means for expressing them. We use "iteration" to describe a step-by-step computational process that determines the next values of a set of state variables from their previous values. A "loop" is a particular control construct, denoted by special syntax, for expressing an iteration, such as Java's `while` and `for` loop constructs. An iteration can also be expressed using tail recursion.)

If loops were easier to learn than recursion, then it would be more difficult to teach a function-oriented language in CS1 than a language with explicit looping constructs. Are people who teach function-oriented languages in CS1 crazy? On the contrary, there is a method to their madness, and we would like to incorporate some of their madness into our methods! In 1997, we completely overhauled our CS1 course, changing from Pascal to Java. As a part of the revamped course, we decided to follow the lead of the function-oriented programming community by experimenting with teaching recursion before loops. Some of us with experience in teaching loops before recursion were skeptical about the wisdom of such a non-traditional approach. However, when we saw

how well the approach worked in practice, the skepticism evaporated. Below, we argue that there are strong pedagogical reasons for teaching recursion before loops in CS1, regardless of the programming language or paradigm employed.

The paper is organized as follows. Section 2 summarizes the status quo in most CS1 courses, where loops are treated as a focus and recursion as an afterthought. Section 3 argues why this situation should be reversed. Section 4 describes our approach to teaching recursion before loops in Java, though the techniques are applicable to any programming language. Section 5 addresses some frequently asked questions about our approach. Section 6 concludes with a summary of our experience in implementing these ideas.

2. THE STATUS QUO

Since we teach Java in CS1, we examined a sampling of CS1 Java textbooks ([ArWe98, Bis97, Cul98, DeDe97, DeHi98, GaMa98, Hor97, LeLo98, WiNa96]) to review their coverage and ordering of recursion and loops. Most of the books ([Bis97, Cul98, GaMa98, Hor97, WiNa96]) treat recursion as an advanced topic that receives only cursory coverage, usually near the end of the book, in a single chapter, and in the context of a few classic examples (e.g., the factorial and Fibonacci functions, the tower of Hanoi puzzle, binary search, mergesort, and quicksort). One [DeHi98] does not mention recursion at all. Only a few ([ArWe98, DeDe97, LeLo98]) cover recursion in more depth and explore recursion examples that are more compelling. In contrast, all the books have significant coverage of `while` and `for` loops, which are typically introduced early and used frequently throughout the remainder of the book. In all books but one, loops were introduced before recursion, and the exception ([Hor97]) covered all but one of its recursion examples after extensive loop coverage.

Based on our experience with introductory C, C++, and Pascal textbooks, the Java textbooks we examined are representative of CS1 textbooks in general. Even as staunch a proponent of recursion as Eric Roberts (see [Rob86]) chooses to discuss recursion only in the very last chapter of his introductory C textbook [Rob95b].

This state of affairs is consistent with the ACM/IEEE task force recommendations for the CS curriculum in a liberal arts setting. For example, [GiTu86] dictates that CS1 "should include all control constructs of a language" (including loops), but takes a weaker stand on recursion: "Recursion should also be given at least cursory treatment even though a full understanding might not take place until CS2." More recent recommendations [WaSc96] are vague, indicating that both iteration and recursion should be covered somewhere in the CS1 and CS2 curriculum.

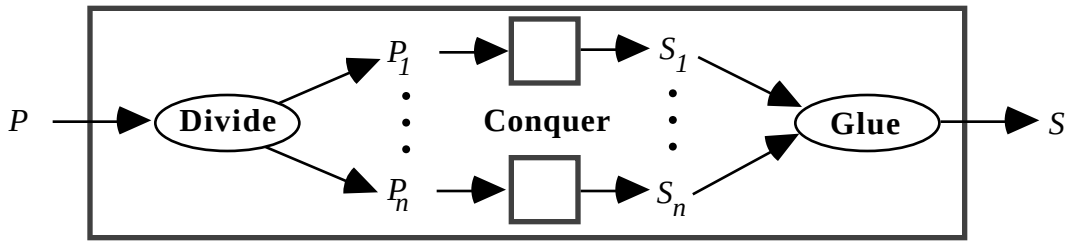
3. WHY TEACH RECURSION BEFORE LOOPS?

Here we argue that there are strong theoretical, practical, and pedagogical reasons to teach recursion before loops.

3.1 Recursion is a Natural Consequence of Divide, Conquer, and Glue

One of the most important ideas taught in CS1 is the *divide, conquer, and glue (DCG)* problem solving strategy:

- *divide* a problem P into subproblems $P_1, \ldots, P_n$;
- *conquer* the subproblems by solving them, yielding subsolutions $S_1, \ldots, S_n$;
- *glue* subsolutions $S_1, \ldots, S_n$ together into the solution S to the whole problem.

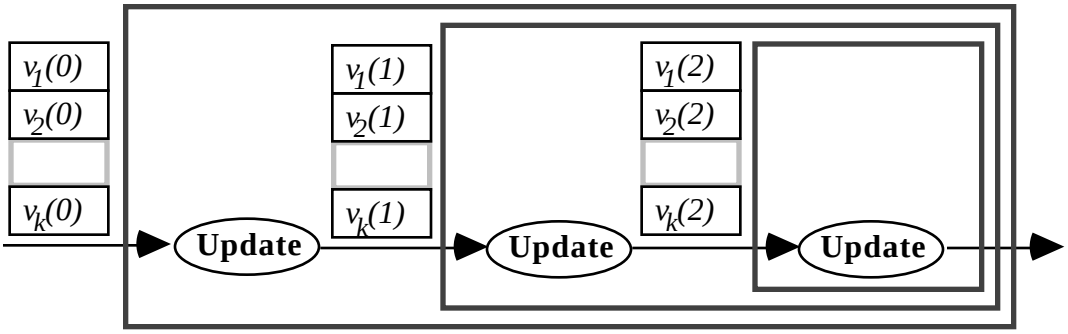


The DCG decomposition is applied until the problems are so small that their solution is trivial. In traditional presentations of this strategy, the "glue" step is implicit, but we think it is very important to make this step explicit because it is a common source of confusion (see Section 3.5) and is also needed to define tail recursion

In CS1, it is standard to teach that functions (or procedures or methods) are the unit of encapsulating the DCG strategy. The input to a function is the problem *P* to be solved; the output of the function is the solution *S* to *P*; and the body of the function performs the divide, conquer, and glue steps. The conquer step is achieved by invoking functions appropriate for solving the subproblems of *P*.

Recursion is an instance of DCG in which some subproblems are the same kind of problems as the original problem. A natural function for solving these subproblems is the one being defined to solve the whole problem. The notion of a function calling itself causes initial confusion for many students, but the fact that it is an instance of a more general pattern with which they are already comfortable helps to resolve this confusion.

Just as recursion can be viewed as an instance of DCG, iteration can be viewed as an instance of recursion in which (1) there is at most one subproblem to be solved and (2) the glue step is trivial (i.e., the identity operator). This pattern of recursion, in which no work is done in the glue step (i.e., between the return of an inner call and the return of an outer call to the recursion) is known as *tail recursion*. If we (1) view the iteration problem as being the values of a set of state variables $v_1(t) \ldots v_k(t)$ parameterized by the time *t*; (2) consider the divide step to update the state variables to their values at time $t+1$ based on their values at time t; and (3) dictate that the glue step is empty, then tail recursion is isomorphic to iteration, as suggested by the following picture:



It is worth noting that the description of the DCG strategy is itself recursive when DCG is used as the subproblem-solving strategy. If we expect students to understand the DCG strategy, we are implicitly expecting them to understand recursion! It is not unreasonable to expect that they should also be able to understand recursive programs.

3.2 Recursion is More Fundamental than Looping

Not only does recursion naturally arise out of DCG, but it is in some sense more fundamental and powerful. It is possible to program comfortably using recursion to the exclusion of loops, but not vice versa. Using loops to encode recursion is more challenging; it requires simulating an explicit call stack, a topic that is normally the domain of CS2, not CS1. There is even a technical sense in which recursion is more powerful than looping. In a language with first-order functions and no data structures, recursion can express computations that are not expressible with loops [PaHe70].

3.3 Tail Recursion is Simpler than Looping

Iterations expressed via tail recursion are often easier to read, write, and reason about than loops. The rigid structure of looping constructs makes it tricky to express iterations that may terminate under multiple conditions, especially if some of the conditions occur in the middle of a loop body or require special finalization actions. As discussed in [Rob95a], such situations are usually addressed by adding extra boolean variables to control the loop or by jumping out of the middle of a loop using a constrained form of **`goto`**. In contrast, tail recursive solutions to these problems tend to be remarkably simple and elegant, because they require none of these special tricks.

For instance, consider a function that returns the index of the first occurrence of a given character in a character array, or -1 if it is not found. Here is a Pascal function that uses a local tail recursive function to express the two separate exit conditions:

```
type Ints = array [0..HI-1] of char;

function search (c:char; var A: Ints): integer;
  function searchLoop (i : integer): integer;
    begin
      if i >= HI then
        searchLoop := -1
      else if c = A[i] then
        searchLoop := i
      else
        searchLoop := searchLoop(i+1)
    end
  begin
    search := searchLoop(0)
  end
```

In contrast, expressing the same iteration in a Pascal **`while`** loop requires introducing a boolean loop control variable that obscures the logic of the iteration:

```
function search (c:char; var A: Ints) : integer;
  begin
    done: boolean := false;
    i: integer := 0;
    search := -1;
    while (not done) and (i < HI) do
      if c = A[i] then
        begin
          done := true;
          search := i
        end
      else
        i := i + 1
  end
```

Some versions of Pascal support a "short-circuit" boolean conjunction operator (**`&`**) that can make the iteration more perspicuous by obviating the need for the `done` variable:

```
function search (c: char; var A: Ints): integer;
  begin
    i: int := 0;
    while (i < HI) & (not (c = A[i])) do
      i = i + 1;
    if (i >= HI) then
      search := -1
    else
      search := i
  end
```

However, when the **`while`** loop exits, an additional test must be performed to see *how* the loop terminated. This test is not implied by the specification of the iteration, but is an artifact of its implementation.

Finally, the iteration can be more succinctly expressed in languages (e.g, C, C++, and Java) with constructs that jump out of the middle of a loop. E.g, in Java:

```
public static int search (char c, char [] A) {
  for (int i = 0; i < A.length; i++)
   {if (c = A[i]) return i;}
  return -1;
}
```

Although the Java solution is very concise, it requires a solid understanding of control flow and the semantics of **`return`**. Also, this solution may be unacceptable to instructors who consider **`goto`** harmful [Rob95a].

Of the above solutions, the tail recursive one best expresses the intrinsic problem-solving logic of the iteration because it avoids the ad hoc hackery required in all of the solutions that use loops. The tail recursive approach is expressible in all general-purpose programming languages, regardless of what non-local exit constructs they support, so it is easier for students to transfer their understanding of tail recursive solutions to other languages they encounter after CS1. Given these advantages, the question is not how to justify teaching iteration via tail recursion but rather how to justify teaching it otherwise!

Although the tail recursive strategy described above works in all languages supporting recursion, there is one fly in the ointment, which we call the *non-block-structure gotcha*. Block structure is a language feature that can significantly simplify the expression of many recursions. In a block structured language, any declaration that can appear at top-level can also appear within the body of a function/procedure/method. In particular, a block structured language allows functions to be declared locally within other functions. Block structured languages include Scheme, ML, and Pascal but not C, C++, or Java. (The inner class feature of Java 1.1 supports a kind of block structure, but may be too complex to introduce to CS1 students.)

Block structure allows a locally declared function to refer to variables of enclosing functions as free variables, rather than requiring them to be passed as explicit arguments to the local function. For example, in the tail recursive Pascal `search` function above, the inner `searchLoop` function need not take `c` and `A` as parameters. However, since Java does not have block structure, the tail recursive approach requires the analogous `searchLoop` static method to take `c` and `A` as explicit parameters, as shown below:

```
public static int search (char c, char [] A) {
  return searchLoop(c,A,0);
}

public static int searchLoop(char c, char [] A, int i) {
  if (i >= A.length)
    return -1;
  else if (c == A[i])
    return i;
  else
    return searchLoop(c, A, i+1);
}
```

(An alternative is to make `c` and `A` instance variables of an object for which `search` and `searchLoop` are instance methods. This "object oriented" approach provides no clear benefit over passing extra parameters but is considerably more verbose and complex.)

Thus, expressing iteration via tail recursion in non-block-structured languages can require passing numerous explicit parameters that would be implicit in the analogous loop. However, we believe that the non-block-structure gotcha is a relatively minor inconvenience when compared to the benefits of tail recursion.

3.4 Recursion Requires Fewer Prerequisites Than Loops

The formal relationship between recursion and iteration has practical import for how CS1 is taught. Assuming that the goal of CS1 is to teach "big ideas", not details, it is desirable to introduce the minimal number of programming constructs that allow the

exploration of these ideas. Recursive programs require only two language features: (1) functions/procedures/methods, which are often covered early in CS1 courses, especially those emphasizing the DCG strategy; and (2) some form of conditional, also typically covered early in CS1. Certainly, (1) is nontrivial. But since both prerequisites are usually taught anyway for other reasons, *no new language features* are necessary to explore recursion or iteration. The execution models for explaining function invocation suffice to explain recursion, so no new models are necessary, either (see Section 4).

In contrast, using loops requires understanding (1) the syntax and semantics of the loop construct(s), as well as (2) the subtleties of assignment. Assignment is not a prerequisite to understanding recursion; in fact, we delay the introduction of assignment until after recursion is taught. (Note that the ":=" in the tail recursive Pascal `search` function above is not a standard variable assignment, but part of Pascal's quirky mechanism for returning a value from a function.) The difficulties of understanding assignment and the flow of control through a loop are often underestimated relative to the difficulty of understanding an activation frame model that explains function calls (and recursion). Understanding loops requires new language details and new conceptual models, yet the resulting tool is nowhere near as powerful or elegant as recursion. In our experience, the notion of updating state variables in a loop is easier to motivate and teach when the concept of iteration is already familiar from tail recursion.

3.5 Avoiding the Iteration Trap

We believe that a key reason that recursion is considered difficult is precisely because it is traditionally taught *after* students have built up preconceptions about self-referential processes based on their experience with looping. As noted above, looping is an instance of DCG in which a problem divides into a single subproblem without a glue step. This lulls students into expecting that all self-referential processes should have these features, and makes it more difficult for them to understand the non-trivial glue steps (*pending operations*) in more general recursive processes.

This "iteration trap" can be circumvented by emphasizing from the start that non-trivial glue steps are a general feature of DCG, not just some quirky property of recursion. I.e., some model explaining pending operations must be used to explain the evaluation of *any* expression that exhibits nesting, whether it exhibits recursion or not.

For example, consider the following mathematical function definitions:

```
p(a) = 1 + f(q(a))
q(b) = 2 * g(r(b))
r(c) = h(c) - 4
```

(This is a contrived example ill-suited for CS1, but space does not permit a more compelling one.) Understanding the evaluation of the function application `p(3)` requires a model that somehow explains the process of unwinding the function definitions, e.g:

```
p(3)
-> 1 + f(q(3))
-> 1 + f(2 * g(r(3)))
-> 1 + f(2 * g(h(3) - 4))
```

The unwinding process leaves behind pending operations (`+`, `*`, `f`, `g`, and `h`) that cannot be performed until their arguments are known. Any model sufficient to explain the dynamic construction and removal of pending operations in the evaluation of non-recursive function calls like `p(3)` is sufficient to explain the evaluation of recursive function calls. Such models include the expression rewriting model shown above as well as activation frame models used to explain the semantics of function calls (e.g. [AbSu95, Rob86]). We believe that many problems students have in understanding recursion can be ameliorated by emphasizing examples that involve pending operations and teaching models that explain pending operations *before* recursion is introduced.

3.6 Deep Ideas Take Time To Absorb

Recursive thinking is one of the deepest ideas taught in CS1. Deep ideas should be introduced early in order to give students time to become familiar with them. The usual

approach of teaching recursion at the tail end of a course seriously short-changes students. They often get just cursory exposure, sometimes without even writing a recursive function on their own. Also, they are typically so saturated with end-of-semester work that it is a struggle to absorb new ideas. Recursion is such an important idea that every CS1 student, regardless of whether or not she is continuing on to CS2, should finish CS1 without a good grasp of recursion. This level of understanding requires time for the idea to germinate and grow.

4. HOW TO TEACH RECURSION BEFORE LOOPS

Since the fall semester of 1997, we have taught recursion before loops in a CS1 course based on Java. This approach works remarkably well. The success of this approach is not just a consequence of the change in topic order, but is due in large part to the way we present the material. In this section, we highlight the aspects of our course that we think contribute to the success of the approach. Although we use Java to teach this approach, there is nothing Java-specific to any of our suggestions.

4.1 Teach Explicit Structural Models

We believe that the single most important factor in teaching recursion before loops is using an explicit execution model that explains function/procedure/method invocations. For our course, we developed a Java Execution Model (JEM) that explains Java method and constructor invocations in a high-level way that abstracts away from any details of computer architecture. The model explains object creation, parameter passing, the allocation, initialization, and assignment of both local and instance variables, the meaning of the Java keyword **`this`**, the execution of the statements in the method body, and the return of a result for methods with a non-**`void`** return type.

The JEM explains the invocation of *any* method, not just recursive ones, so we introduce it as part of understanding methods. In particular, the JEM explains pending operations, a notion essential for understanding recursion. Empirical studies confirm the importance of execution models for students learning recursion [WDB98].

In our review of Java textbooks, we were flabbergasted by how few present an explicit execution model of method invocation. Those that do often wait until recursion to introduce such a model, which is too late. An execution model is necessary not only to understand recursion, but to understand many other subtle aspects of Java execution. Students must have *some* model to understand execution, and if you do not teach them one, they will invent one (or several) on their own. The problem is that the models they intuit are not likely to handle all of the subtleties of execution correctly. A host of problems can be avoided by teaching such a model explicitly.

4.2 Emphasize Divide, Conquer, and Glue (DCG)

It is difficult for students to see the connection between DCG and recursion if it is not taught explicitly. When we show how Java methods can be used to decompose problems, we emphasize how methods are the embodiment of the DCG strategy. This paves the way for motivating recursion as an instance of DCG.

4.3 Use Recursion Early and Often

Many CS1 books have a single chapter on recursion, and do not use recursion after this chapter. This compartmentalized approach to recursion suggests that it is an optional topic that is peripheral to the goals of CS1. We believe recursion is integral to CS1, and must be taught in a way that emphasizes its central importance. We introduce recursion in four lectures beginning with the 13th lecture in a 26-lecture course. Many of the previous lectures focus on methods and their relationship to the DCG problem solving strategy (4 lectures) and conditionals (2 lectures). Loops are introduced in the two lectures following the four recursion lectures. Students are given two homework assignments on recursion and one on loops (details below).

Both recursion and loops are used often in the lectures and assignments in the remainder of the course. For example, array and linked list algorithms are expressed using both recursion and loops. This emphasizes that recursion is a fundamental technique that is useful beyond the examples in which it is introduced.

4.4 Just Say No to Factorial

In our course, we develop students' interest in recursion by showcasing it in examples that are more compelling than the traditional examples of factorial and Fibonacci. Specifically, we introduce recursion in the context of three "microworlds" that are already familiar to the students from their study of methods and conditionals:

- *BuggleWorld* is a microworld, inspired by Karel the Robot [BSRP97], in which "creatures" known as *buggles* populate a grid of cells. Each buggle is characterized by its position, heading, color, and the state of a pen (up/down) that leaves a mark when it moves. Buggles can turn and move from cell to cell, except when they are facing a wall, which they can detect. They can also sense, pick up, and drop their favorite food: bagels. Whereas [BSRP97] considers recursion to be an advanced topic that is not covered until the final chapter, we teach recursion with buggles early on using examples that students find motivating (see the next subsection).
- *TurtleWorld* is a Java implementation of the turtle graphics microworld pioneered in LOGO [Pap80]. Recursion examples include drawing polygons, spirals, grids, and self-similar fractals like trees, snowflakes, and Sierpinski's gasket [PJS92].
- *PictureWorld* is a coordinateless graphics microworld inspired by the Escher picture language used in [AbSu95]. In this microworld, complex pictures can be expressed by transforming (e.g., scaling, rotating, flipping) and combining (e.g., overlaying, vertically and horizontally juxtaposing) simple shapes (e.g. lines, triangles, rectangles). Recursion examples include recursive quilt patterns, archery targets, and pictures of binary trees.

Such graphical microworlds have two advantages over traditional examples as vehicles for introducing recursion. First, students find them more exciting. Second, since each action gives visual feedback, incorrect recursive methods are easier to debug.

Examples from the above microworlds are used in the first three lectures and two labs on recursion. Only in the very last of four lectures on recursion do we cover classic examples of recursion like factorial, Fibonacci, and the towers of Hanoi. We do this partially for cultural reasons and partially as an means for introducing static methods, which have not been used earlier in the course.

4.5 Pedagogical Progression

The microworlds mentioned above are natural contexts in which to implement a pedagogical sequence based on DCG that culminates in recursion and iteration. Below we describe the pedagogical progression leading to recursion in terms of BuggleWorld examples, though we also use TurtleWorld and PictureWorld examples at every stage.

- Students are introduced to buggles via simple programs that exercise their capabilities. To accomplish tasks, students write straight-line programs in the body of a single distinguished `run()` method in a subclass of the `BuggleWorld` class.
- Methods are introduced as the tool for decomposing buggle tasks according to the DCG strategy. The JEM is introduced as a way to explain method invocation, especially the concept of pending operations. Students experiment with adding methods (but not instance variables) to numerous subclasses of the `Buggle` class.
- Conditionals are introduced as a way to programmatically use a buggle's sensors. They are used within a buggle `step()` method that is invoked some large but fixed number of times in order to accomplish tasks involving the traversal of many grid cells. In this "pseudo-loop", the buggle is programmed to do nothing within the `step()` method once it reaches a desired state. Tasks include dropping bagels in every grid cell, following a bagel trail, and finding a bagel in an acyclic maze.

- Dissatisfied by the inelegance of the pseudo-loop, students are ready to learn a mechanism in which the buggle performs no more steps than necessary in order to accomplish a task. This mechanism is recursion! Examples at this stage include tail recursive processes (e.g., leave a trail of bagels to the wall, walk forward until finding a bagel or wall) as well as non-tail recursive processes in which pending operations are used to undo certain changes to the buggle's state (e.g., leave a trail of bagels to the wall and return to the initial position and heading, jump over a wall of unknown height, find a bagel in an acyclic maze and return to the starting position). We illustrate both tail recursive and non-tail recursive methods in the very first recursion lecture to help avoid the iteration trap. We also use the already familiar JEM to explain the pending operations in the non-tail recursive case. A key aspect of this stage is that the previously learned execution model completely explains all aspects of recursion without any modifications.
- The next stage focuses on the return of values from recursive buggle methods. Examples include determining if there are any bagels in the line of sight of the buggle and determining the number of steps to the wall the buggle is facing. In each of these examples, the final position and heading of the buggle should be the same as the initial position and heading. The solutions to these problems have general cases with the following form:

  To return the value for a buggle in state S:
  1. Change the state of the buggle from S to S';
  2. Determine the result R' of recursively invoking the method at S';
  3. Undo the state change in 1, from S' back to S;
  4. Perform any glue on R' to produce the final result R, which is returned.

  These sorts of recursions are far more complex than those typically covered in a CS1 course. Indeed, there are several pitfalls for students (e.g., forgetting to name R' to communicate it from step 2 to step 4; neglecting to undo the state change in step 3). We have found it important to teach the above idiom explicitly, to present many examples of this idiom, and to use the JEM to show in detail how some of the examples work.

  Note that we do *not* introduce instance variables, or even assignment to local variables, before we cover recursion. This narrows the space of possible solutions by precluding solutions that store the result of the recursion in a buggle instance variable. This forces students to understand the idiom of returning recursive values without relying on instance variables as a crutch. When instance variables are introduced later, we explain how instance variables provide an alternative solution to this class of problems. Delaying the introduction of assignment and instance variables is another influence of function-oriented programming on our course.
- Iteration is introduced as a means of calculating one row of a two dimensional table from the previous row. Each column of the table represents the values in a particular state variable over time, and each row represents the values of all state variables at a particular point in time. The connection between iteration and tail recursion is made: any iteration can be expressed as a tail recursive method that has one parameter for each state variable. Buggle tail recursions are viewed as iterations on state variables.
- **`while`** and **`for`** loops are introduced as a concise way of expressing common patterns of iteration. It is now necessary to introduce assignment so that the value of a variable may change during the execution of a loop body. A correspondence is made between the values of the loop state variables at a particular point in time and the parameters to a particular invocation of the corresponding tail recursive method. Familiar buggle tail recursions are re-expressed using loops.
- Later in the course, arrays and linked list structures are presented and are manipulated via recursive methods (tail recursive and non-tail recursive) and loops. Recursive data structures are normally not introduced until CS2, but teaching simple ones in CS1 dovetails nicely with our approach of teaching recursion before loops. Programs that build and/or traverse recursive data structures are perhaps the most

compelling and practical use of recursion, so it is important to see some examples in CS1. In fact, in the future we plan to experiment with teaching binary trees in CS1, because their branching structure leads to patterns of recursion that are not encountered with linear lists and are difficult to express via iteration.

5. FREQUENTLY ASKED QUESTIONS

Here we answer some questions that are, or we expect to be, asked about our approach. It is worth noting we raised many of these questions ourselves before teaching the approach advocated by this paper.

Isn't recursion too complex and advanced to learn early in CS1?

Ask this question of the thousands of grade schoolers who have successfully written recursive turtle programs in Logo [Pap80]. If grade school students can routinely learn recursion, it is not too much to expect of college students! This is not to say that recursion is simple to learn. There are many challenges and potential pitfalls. But it is our experience that most students can work through these difficulties if they are taught appropriate models and motivating examples. Finally, we contend that the "recursion is hard" belief is partially a cultural phenomenon that is perpetuated by instructors and textbook writers who were themselves indoctrinated with this belief. If students are told enough times that something is difficult, then it is not surprising that they will believe that it is difficult.

Aren't loops easier to read/write than recursions?

A main reason loops are considered to be simpler than recursion is that they can be introduced via a few standard idioms, such as the "repeat N times" idiom in C and Java:

```
for (int i = 0; i < N; i++) {
   statements that use i
}
```

Students can often effectively use such idioms without necessarily understanding all of the details of the looping construct.

However, similar idioms can be taught in a recursive style. The above example can be expressed via a tail recursion:

```
... repeatMethod (0) ...

void repeatMethod (int i) {
  if (i < N) {
    statements that use i;
    repeatMethod(i + 1);
  }
}
```

It is true that looping constructs are often more concise than their recursive analogs, that their organization highlights aspects of the iteration better than the recursive idiom, and that they avoid the non-block-structured gotcha (see Section 3.3). These are all reasons why we do eventually teach looping constructs in our course.

There are too many topics in CS1 already. How can I devote the amount of time to recursion that your approach requires?

It is difficult to emphasize how important recursive thinking is to successfully navigating the rest of the computer science curriculum. We believe that allocating time and energy to teaching recursion early can have handsome payoffs in later courses.

It is our experience that much time in traditional CS1 courses is spent on secondary details rather than primary ideas. It is not necessary to study every control construct, every data structure, and every primitive operator supplied by a language. Features should only be presented to the extent that they help teach a big idea. For example, conditionals are a big idea. In Java, you can express all conditionals with the **`if`**

construct. Conditionals can also be expressed via the **`switch`** construct. We feel the added detail of the **`switch`** statement is unimportant. The time it takes to teach this can be better spent on other big ideas, like recursion.

Many CS2 courses have significant coverage of recursion. Some time can be gained in CS1 by swapping CS1 material with the CS2 recursion material.

Doesn't encouraging recursion promote inefficient programming?

In many languages, recursion, even tail recursion, implies significant time and space overheads relative to looping constructs. However, this sort of inefficiency does not concern us for three reasons: (1) While we teach recursion before loops, we still do teach loops, so students eventually use them in their programs. (2) We believe that premature emphasis on efficiency in CS1 is detrimental to the spirit of experimenting with various approaches to solving a problem. Too often students ignore elegant solutions or focus on irrelevant details because they are overly concerned about efficiency. We believe that efficiency is an appropriate focus in subsequent courses, but not in CS1. (3) Implementations of recursion are not necessarily inefficient. It is possible to implement tail recursion so that it executes precisely like a loop [Ste77]; in fact, Scheme implementations are *required* to have this behavior. By training more students to understand and use tail recursion, we hope to encourage future language designers and implementers to support efficient tail recursive calls in *all* programming languages.

Aren't languages like Lisp, Logo , and ML better for teaching recursion than Pascal, C, C++, Java, etc.?

Modulo the non-block-structured gotcha (see Section 3.3), any recursive algorithm can be expressed in any of these languages in almost exactly the same way. Many dialects of Lisp provide imperative looping constructs, so the proclivity of Lisp programmers to use recursion is more a matter of culture than of language. We hope that teaching recursion early will help to spread the elegant programming aesthetics of the functional programming culture to other programming language communities.

If Lisp (and similar languages) can be considered to have any advantages relative to recursion they are that (1) it is block structured, and so avoids the non-block-structured gotcha; (2) some dialects (namely Scheme) require that all implementations execute tail recursive function calls as efficiently as loops; and (3) function-oriented languages are typically garbage collected, which makes recursive functions that take or return dynamically allocated structures like lists and trees more convenient, since these structures need not be manually deallocated. Advantage (1) is shared by many languages that are not function-oriented, e.g. Pascal and its descendants. Note that Java already has feature (3) and may one day have feature (2).

6. EXPERIENCE

It is our experience that introducing recursion before loops as sketched above is a preferable alternative to the traditional approach. Judging by their office hour visits and their performance on problem sets and exams, we believe that most of our students finish the course with a firm understanding of both recursion and loops. Some students still harbor confusion about recursion at the end of the course. The kind of example most likely to still be confusing is a recursion that involves both returning a value and undoing a state change.

We have observed that students seem to leave our CS1 course with better problem-solving skills than in the previous incarnation of CS1. We believe that this is largely due to the fact that we now place much more emphasis on the divide, conquer, and glue strategy. Explicitly viewing recursion and iteration as instances of this strategy dovetails nicely with this emphasis. Students see that recursion and iteration are not ad hoc techniques, but examples of already familiar principles.

The fact that no existing CS1 textbook (at least in Java) teaches recursion before loops or even has extensive coverage of recursion is a barrier to adopting our approach. Since we have not found an adequate textbook, we expect students to rely on notes they take in lecture for learning the material on the Java Execution Model, DCG, recursion, and iteration. Eventually, we hope to produce on-line notes that we can share with the community at large.

The lack of textbook also implies a lack of readily available examples. In addition to implementing the microworlds described above, we have developed a series of examples and exercises based on these microworlds. We encourage other members of the community to use these microworlds and examples (with proper attribution, of course). Please contact us or visit the following web site:

`http://nike.wellesley.edu/~cs111/it's-nice-to-share.html`

We strongly encourage other CS1 instructors to experiment with teaching recursion before loops, and to share their experiences with the CS1 community.

ACKNOWLEDGMENTS

We thank Allen Downey and the anonymous reviewers for their helpful comments.

REFERENCES

[AbSu95] Abelson, Harold, and Sussman, Gerald, with Sussman, Julie. *Structure and Interpretation of Computer Programs*. MIT Press, 1995.

[ArWe98] Arnow, David, and Weiss, Gerald. J*ava: An Object-Oriented Approach*. Addison-Wesley, 1998.

[Bir98] Bird, Richard. *Introduction to Functional Programming Using Haskell.* Prentice Hall Europe, 1998.

[Bis97] Bishop, Judy. *Java Gently: Programming Principles Explained*. Addison/Wesley/Longman, 1997.

[BSRP97] Bergin, Joseph, Stehlik, Mark, Roberts, Jim, and Pattis, Richard *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming.* John Wiley and Sons, Inc., 1997.

[Cul98] Culwin, Fintan. *Java: An Object First Approach*. Prentice Hall, 1998.

[DeDe97] Deitel, H.M., and Deitel, P.J. *Java: How to Program*. Prentice Hall, 1997.

[DeHi98] Decker, Rick and Hirshfield, Stuart. programming.java: *An Introduction to Programming Using Java*. PWS, 1998.

[GaMa98] Garside, Roger, and Mariani, John. *Java: First Contact.* Course Technology, 1998.

[GiTu86] Gibbs, Norman E., and Tucker, Allen B. "A Model Curriculum for a Liberal Arts Degree in Computer Science." *CACM*, 29(3), March 1986. Pp. 202--210.

[Hor97] Horstmann, Cay S. *Computing Concepts with Java Essentials*. John Wiley & Sons, Inc, 1997

[LeLo98] Lewis, John, and Loftus, William. *Java: Software Solutions*. Addison-Wesley, 1998.

[Pap80] Papert, Seymour. *Mindstorms.* Basic Books, 1980.

[PaHe70] Paterson, Michael S. and Hewitt, Carl E. Comparative Schematology. *Proceedings of ACM Conference on Concurrent Systems and Parallel Computation,* 1970. Pp 119-127.

[Pau96] Paulson, L.C. *ML for the Working Programmer*. Cambridge University Press, 1996.

[PJS92] Peitgen, Heinz-Otto, Jurgens, Hartmut, and Saupe, Dietmar. *Fractals for the Classroom, Part One: Introduction to Fractals and Chaos.* Springer Verlag, 1992.

[Rob86] Roberts, Eric S. *Thinking Recursively*. John Wiley, 1986.

[Rob95a] Roberts, Eric S. Loop Exits and Structured Programming: Reopening the Debate. In the *26th SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* 27(1), March 1995. Pp. 268--272.

[Rob95b] Roberts, Eric S. *The Art and Science of C: A Library-Based Introduction to Computer Science*. Addison-Wesley, 1995.

[Ste77] Steele, Guy L. Debunking the "Expensive Procedure Call Myth", or, Procedure Call Implementations Considered Harmful, or LAMBDA the Ultimate Goto. MIT *Artificial Intelligence Laboratory Memo AIM-443*, Oct. 1977.

[WaSc96] Walker, Henry M., and Schneider, Michael G. "A Revised Model Curriculum for a Liberal Arts Degree in Computer Science." *CACM*. 39(12), Dec. 1996. Pp. 85--95.

[WDB98] Wu, Cheng-Chih, Dale, Nell B., and Bethel, Lowell J. Conceptual Models and Cognitive Learning Styles in Teaching Recursion. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)*. 30(1), Mar. 1998. Pp. 292-296.

[WiNa96] Winston, Patrick , and Narasimhan, Sundar. *On To Java*. MIT Press, 1996.