Self-Reference is an Illustrative Essential

Owen Astrachan
Department of Computer Science
Duke University
Durham, NC 27708
ola@cs.duke.edu

Abstract

This paper includes an abstract, a discussion of the usefulness of self-reference in early computer science courses, and some examples to illustrate this usefulness. Most readers will not be troubled by this example of self-referential writing. Why then is self-reference, usually in the form of recursive subprograms, thought to be so onerous as to be placed in its own left-until-the-end-and-often-uncovered chapter in most introductory texts? Self-reference is one of the cornerstones of computer science from the unsolvability of the halting problem, to writing a Pascal compiler in Pascal, to reveling in the beauty of Quicksort. We argue that the notion of self-reference should permeate first courses in computer science. If this is to be the case such courses should take a view far broader than "Wow, I can average 10 numbers with the skills I learned in my first programming course!"

1 Introduction

Students in a first course in computer science have difficulty with a wide variety of topics. Although computers are ubiquitous at all educational levels, many students often have trouble understanding what an operating system is, what a compiler does, why it is hard to program, and what a computer can and cannot do. Although we have no panacea to cure all these ills, we argue in this paper that the general concept of *self-reference* is not only of fundamental importance in computer science, but is useful in helping students come to understand many basic concepts of computer programming and computer science.

We offer several examples of self-reference that can be used to explain different aspects of computer science. Most of the examples occur in the context of programming since we believe that programming on a real machine should be an integral part of introductory courses and that whenever possible advanced topics should be introduced and developed in a programming context. We present examples in either C++ or Pascal, but any imperative language could be as easily used. Many of our examples are also useful in first courses that use Scheme, but by design recursion in these courses plays a different role than it does in courses based on Pascal-like[1] languages. We have used these examples successfully in survey courses for non-majors as well as in our first course for majors. In general, student enthusiasm for these examples runs high.

Before developing our examples, we offer support for our position from Daniel McCracken, the 1992 winner of the SIGCSE award for contributions to computer science education [McC87]:

Is Recursion an Advanced Topic?

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. It is too important and too valuable to be belittled by showing a recursive factorial function in CS1, which conveys almost nothing of its power.

Recursion is not trivial, but it is not tensor calculus.

We owe it to our students to treat recursion and self-reference as an accessible topic and not to stamp it with a "badge of dishonor". By covering recursion early, for example, students may come to understand it before learning enough to know that it's difficult.

[1] We include Pascal, C, C++, Modula-2, and Ada in this list although each of these differs from the others in fundamental ways.

2 The First Example

Students study some syntactic details of a language in order to be able to write their first program. In [Pat94], Rich Pattis argues that studying syntax in the form of EBNF grammars provides a productive beginning to introductory courses. Such grammars invariably include self-referential rules for expressions and identifiers, providing an early use of self-reference that is accessible to students.

Many courses and texts begin the study of a language using programs whose only task is to write strings; the canonical example of such a program is one that prints "Hello, World" [KR78]. Such programs are useful because students must be able to use an editor, a compiler, and understand rudimentary features of an operating system in order to be able to run them. In our first course we use such programs as well, but the programs print a sequence of characters that is a superset of "Hello World". The program and the output it produces are illustrated in Figure 1.

```
program MetaHello;
begin
    writeln('program HelloWorld;');
    writeln('begin');
    writeln('    writeln(''Hello World'');');
    writeln('end.');
end.
```

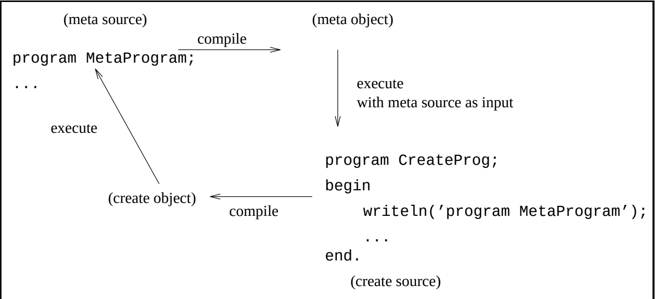
```
program HelloWorld;
begin
    writeln('Hello World');
end.
```

Figure 1: An interesting first program

We ask our students to save the output in a file and to compile and execute it. When students understand everything they have done with this example, they have a much deeper understanding of programming and computers than they do using a traditional "Hello World" program.

It is a very small step to generalize *metahello.p* to one that prompts the user for a string and then generates a program that, if compiled and run, will print the string as output. Thus the user/writer of the program has, to a degree, parameterized *metahello.p* by increasing its functionality from a "Hello World" generating program to an any-string generating program. Students can easily construct such a program and then generalize yet again to a program that prompts for a file for input rather than a string and then generates a program that, if compiled and run, will print the file as output. We provide students with such a program and have them experiment by running the program on *metahello.p*. This produces a program whose output is a program whose output is a program. Although this is initially very confusing for students, they can examine the output from each stage of this process in coming to grips with the process of compiling and running programs.

Finally, to emphasize the self-referential aspects of this problem even more, students run the file-printing meta-program (object) on itself (source); this process is illustrated in Figure 2.



```
program MetaProgram;
...
```

```
program CreateProg;
begin
    writeln('program MetaProgram');
    ...
end.
```

Figure 2: a (meta) program that creates programs

Thus, in some sense, self-reference has come full circle. In the process we have discussed compilers, I/O, redirection (if using a DOS or Unix[2] environment), and both source and object files. Students come to understand the last distinction since running the last program on the object version of itself produces a much different program than when run on the source version of itself. The idea of *parameterization* is also emphasized since students use a program that produces output that differs according to the input.

At the same time this exercise makes it clear that running programs on other programs is an every-day occurrence since compilers are certainly programs, editors are programs, and students have now used another program that accepts programs as input. We also provide a degree of foreshadowing by running a program on itself; we return to this later in discussion of noncomputability and the halting problem.

Student enthusiasm for the "first program" is much higher since we have incorporated these ideas rather than relying on such typical examples as adding two numbers or computing interest rates. Although this is anecdotal evidence, we have found that a higher degree of enthusiasm usually translates directly to better performance.

3 Recursive Programs

Almost all introductory texts, regardless of the imperative language being covered, treat recursion as a "special" topic. The canonical examples of factorial and Fibonacci numbers are often covered with only some texts

[2]Unix is a registered trademark

advising that recursion is not the "right way" to solve these problems. Students are left to ponder exactly what recursion is good for since they see few examples of it and the examples are not integrated throughout the text.

Our philosophy is that students should be shown recursive subprograms whenever appropriate. Beginning students should be asked to modify recursive subprograms to show an understanding of how recursion works. Sometimes it may be useful to require certain subprograms to be written recursively, but it is only when a student realizes on her own that recursion is the appropriate tool that the tool is fully appreciated.

Most importantly, however, students should be shown as many examples as possible in order for them to come to "believe" in recursion. Rather than adopt an operational viewpoint and diagram the state of the stack over a series of recursive calls, we believe that the beauty and power of recursion are best understood at a high level by the use of concrete examples. Only after students have begun to understand conceptually how recursive subroutines work at a higher level should they be shown details of how recursion is implemented.

3.1 Writing Backwards

One of the early exercises we use in our first course for majors[3] requires students to calculate the average word length in texts by authors such as Melville, Twain, and Poe. On many machines the intermediate calculations cause integer overflow leading to "interesting" results. This leads to an informal discussion of how integers are implemented. We do not discuss this at the level of explaining *two's complement*, but we do expect students to understand the binary representation of an integer. We use the routine shown below to print a number. Students are asked to reason about switching the order of the statements in the body of the if statement and to experiment with printing different numbers in binary (for example, what happens when the precondition is violated?).

```
void
PrintBinary(int n)
// precondition:  n > 0
// postcondition: n printed in binary
{
    if (n != 0){
        cout << (n % 2);
        PrintBinary(n / 2);
    }
}
```

The study of this problem leads to a similar exercise that we have used successfully in developing an understanding of recursion: printing a number with commas inserted properly, e.g., 31,415,926. If students are given code that is tail-recursive (as in the binary number example above) so that it fails to work as intended, they can usually determine (if only by trial and error) how to fix the code so that it performs properly.[4]

3.2 Towers of Hanoi

Programs that compute the moves necessary to solve the Towers of Hanoi problem are often used to introduce recursion. It is a simple matter to bring an actual representation of a six disk Towers of Hanoi to class and discuss the problem. In every semester in which we have done this, a student develops a recursive formulation of the solution, usually in conjunction with showing how many moves are necessary to move n disks.

Unfortunately, not enough use is made of this problem in most courses. In addition to providing an example of recursion that can be readily understood, the Towers of Hanoi is an example of a provably intractable (i.e., exponential) problem. It is very important for students in the first courses in computer science to have a rudimentary understanding of algorithmic complexity. In addition to understanding the difference between $O(n \log n)$ and $O(n^2)$ sorting algorithms, students should understand the difference between an NP-complete problem (most likely an intractable problem) and a provably intractable problem. Students should also be aware that such asymptotic measures are important, but may not be relevant if the problem size is small.

To introduce these concepts in our non-major's class, we use a new measure of computer performance abbreviated DIPS, for <u>D</u>isks <u>P</u>er <u>S</u>econd, that represents the number of disks moved per second in solving, for example a 25 disk Towers of Hanoi. Our students modify a program that prints the moves so that it counts the moves instead thus avoiding measuring I/O. Table 1 indicates DIPS performance for the machines in our department (25 disks requires $2^{25} - 1 = 33,554,431$ moves.)

machine	time for 25 disks	DIPS
386SX	46 minutes	12,000
486 (33 Mhz)	10.5 minutes	53,000
convex C1	4.6 minutes	120,000
dec 2100	85 secs	390,000
dec 3100	63 secs	530,000
sparc 1+	61 secs	550,000
sparc 2	26 secs	1,250,000
dec alpha AXP	9 secs	3,700,000

Table 1: Measuring Performance using DIPS

This example of recursion has allowed us to talk of complexity classes and performance measures. It is particularly enlightening since students can see some of the

[3]This course recently switched from a C based course to a C++ based course.

[4]This problem appeared on the 1989 Advanced Placement Computer Science Exam.

evolution of performance by their own experimentation. Of course the same results can be obtained by using fewer disks without waiting for nearly an hour. This forces the students to think about testing since the DIPS measure is independent of the number of disks (provided enough moves are made to allow timing to take place.)

We also ask students to try to "improve" the DIPS rate. Since the number of disks moved cannot change, several possibilities arise. Students easily discover that the overhead of procedure calls affects the performance and move the statement that increments the number of moves so that it occurs inline rather than as part of a *MoveOneDisk* subprogram. Occasionally students will have seen this problem before and know that an iterative solution is possible. Although more complex to code, such a solution does improve the DIPS rate. This further illustrates exactly what recursion often means: simplicity and elegance at (perhaps) the expense of performance.

3.3 Non-trivial Recursion

Although we introduce what are considered trivial (e.g., tail recursive) examples of recursion as conceptual illustrations, we strongly believe in studying simple, but useful and powerful examples of recursion. Space considerations preclude a detailed discussion of Quicksort or Mergesort, but either of these sorting routines is accessible to students in a first course. Quicksort in particular can be clearly written in fewer than 20 lines of code.

Removing Directories

In some of our classes students use machines that run a variant of the MS-DOS operating system. This system provides no method for removing a directory when the directory is not empty. Since the remove command does not work hierarchically (recursively), e.g., it cannot be invoked to remove a directory and all its files and subdirectories, disk re-organization can be a time-consuming task without some other support.

Since modern file systems are hierarchical they readily lend themselves to recursive examples. We give our students the code for procedure *Visit* in Figure 3 that prints all files and subdirectories in the directory specified by the initial string parameter.

All procedure calls used in this code are implemented in a unit linked in when this procedure is compiled. This allows us to use the same code across different platforms and operating systems[5]

Our students are asked to reason about how to change this code so that it can be used to remove a directory (in-

```
procedure Visit(s : string);
var
    dirInfo : DirInfoType;
begin
    GetFirstDirEntry(s,dirInfo);
    writeln('DIR: ',DirName(dirInfo));

    while MoreFiles(dirInfo) do begin
        if IsDirectory(dirInfo) then
            if IsRealDirectory(dirInfo) then
                Visit(FullPathString(dirInfo))
        else
            writeln('  ',FileName(dirInfo));
        GetNextDirEntry(dirInfo);
    end;
end;
```

Figure 3: A useful recursive procedure

cluding all subdirectories). This entails understanding not only that the *writeln* statements must be changed, but that the (now changed) first *writeln* must be moved after the loop to ensure that the directory is empty when the (now changed) *writeln* is executed. Again a real conceptual understanding of the process is necessary in order to realize that the removal of the directory must occur after the recursive calls have finished.

Program testing, introduced in most courses but often unmotivated, is very important using this example. How should the modified *Visit* procedure (that removes directories) be tested? Students that fail to talk proper precautions when testing can quickly learn that care is necessary when testing programs that might cause "unwanted side-effects."

4 Noncomputability

Too often students in a first class are left with the impression that the computer is capable of performing almost any task provided an intelligent programmer is at work and enough computer time is available. We believe it is important for students to see that there are problems that are provably unsolvable by computer. The canonical example of a such a problem is the *halting problem*: does a program P halt on input S? In our development of this problem we restrict programs to those whose only input is a string. The question we pose to our students is "Is it possible to write a program corresponding to the machine on the left in Figure 4?" For a similar development see [GL88].

We construct a program *Confuse* as represented by the machine on the right in Figure 4 and predicated on the existence of the program *Halt*. We pose the following three questions to our students and we leave them to the audience of this paper.

1. Describe what it means when *Halt* outputs "yes"

[5]We use the Pascal code in our non-majors course and both C and C++ code in our courses for majors. The code will be provided on request via email.

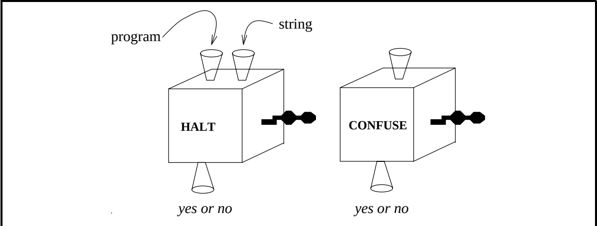


Figure 4: halting problem

when run using the program *Confuse* as the program input and the string representing the program *Confuse* as the string input.

2. Describe what it means when *Halt* outputs “no” with the same inputs as in the previous question.

3. Based on the answers to the previous questions and the semantics of *Confuse*, what is output when *Confuse* is run using *Confuse* as input?

Note that we have again come full circle and returned to a characterization of the first programs we studied: we are running a program on itself as input. Although our students are initially confused by this example (the name *Confuse* is not an accident) we show them a familiar paradox:

This sentence is false.

at the same time, the idea behind the program is not completely unfamiliar because of the self-referential programs they have seen throughout the semester. Just as students have difficulty with *reductio ad absurdum* proofs, self-referential programs can be initially confusing. If such programs are used throughout a course rather than as a “special topic”, students often don’t know that they should be confused and hence take to the material more readily.

```
program Confuse;
var
    s : string;
begin
    readln(s);
    if Halt(s,s) = 'yes' then
        while 0=0 do
            (* loop forever *)
    writeln('yes');
end.
```

Figure 5: the program *Confuse*

5 Conclusion

We believe that the idea of self-reference is important and that it should be a recurring theme in any first course in computer science. We strive to use self-referential examples and recursive procedures in a manner that illustrates their power and that allows us to introduce fundamentally important topics in computer science at a lower level than is normally expected.

We want our first courses to use programming as a means of exploring advanced topics and to use self-reference as one of the themes that ties together the seemingly disparate areas that comprise computer science into a coherent whole rather than leave our students with the impression that our field is comprised of an eclectic collection if disconnected subfields. Although most books do not integrate recursion into the text as a whole, the books by Roberts [Rob86] and Rohl [Roh84] provide many examples of recursion in interesting settings.

Students in our courses for majors do not have as much difficulty with recursion since we have incorporated self-reference into these courses as a theme rather than as a special topic. In addition, they have a better understanding of what computer science is. Finally, students enjoy these examples which usually means that they are willing to work harder at understanding them (and the material in the rest of the course). When asked to comment on those parts of their course that were good, many students invariably choose the study of one of the recursive and self-referential examples we have examined.

Acknowledgments

This paper benefited greatly from comments made by John Rager. Paraphrasing Stuart Reges, I’ve benefited greatly from many of Rich Pattis’ ideas and comments as well.

References

[GL88] L. Goldshlager and A. Lister. *Computer Science: A Modern Introduction.* Prentic-Hall International, second edition, 1988.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[McC87] D.D. McCracken. Ruminations on computer science curricula. *Communications of the ACM*, 30(1):3–5, January 1987.

[Pat94] Richard E. Pattis. Teaching ebnf first in cs 1. In *The Papers of the Twenty-Fifth Technical Symposium on Computer Science Eduction.* ACM Press, March 1994. Phoenix, AZ.

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

[Roh84] J.S. Rohl. *Recursion via Pascal.* Cambridge University Press, 1984.