Explaining Recursion to the Unsophisticated

S. M. Haynes
Department of Computer Science
Eastern Michigan University
Ypsilanti, Michigan 48197
haynes@emunix.emich.edu

Abstract

This paper addresses the topic of explaining recursion to beginning programmers. It briefly presents the common approaches, then describes an extension to those methods called the *activation tree*.

1. The Problem

Recursion is a difficult concept for beginning programmers though its importance, even in the early stages of the computer science curriculum, is generally recognized [Ast94], [McC87]. Recursion is important and subtle; explaining it to a naïve audience is challenging. Beginning programmers tend to get lost in a jungle of detail when tracing through recursive programs. Yet, being beginners, they are not yet ready for the simplifying abstractions in the expert's kit.

I have used all five of the ways of explaining recursion described in the following section. The first four are commonly used approaches. To this arsenal, I've added the *activation tree*, a combination of certain information in runtime stack frames, and the topology of the recursion tree. Students have found it to be particularly illustrative, as it gives them control over all the necessary detail.

This short paper very briefly describes the most common approaches to explaining recursion: induction, runtime stack, the trace, and the recursion tree. It then describes the activation tree, showing how it relates to the common methods. An example of a complicated recursive program, which is traced using the activation tree and the trace, is then given. Using a simpler example, I make some remarks on using the activation tree to help the students improve their understanding of the concept of induction.

2. Ways of Explaining Recursion

1. Induction

The inductive approach gives the high level function definition, i.e., how the function is defined in terms of itself and base case. The EBNF definition of expressions is one such example. One inductive definition of the Fibonnaci series is:

Fib(k) = Fib(k-1) + Fib(k-2);
Fib(2) = 1;
Fib(1) = 1;

The power of this approach cannot be underestimated. However, students frequently can understand the words without appreciating the meaning. Understanding induction is the expert-level abstraction alluded to in the first section.

2. Runtime stack

Recursion is implemented in high level languages by pushing and popping stack frames, or activation record instances, onto the the runtime stack. Each time a procedure is invoked, a stack frame, containing local variables, parameters, return address, and other bookkeeping information is pushed onto the runtime stack. When a called procedure returns control to its calling procedure, the stack frame for the called procedure is popped off the runtime stack.

The advantage of using the runtime stack to explain recursion is that the stack frames keep

track of all information required: parameter values, local variables, return address, and returned value.

The problem with the runtime stack is that students necessarily are taking notes on the static medium of paper, while the runtime stack is a dynamic phenomenon. What with all the pushes and pops, their notes become illegible and useless.

3. The trace

Another technique is the trace. Every time a procedure is called, a line with procedure name and input parameters are printed. Every time a procedure completes execution, a line with procedure name and return value is printed out. The output is indented so that corresponding invocation and return are aligned.

The trace can be thought of as a static (pen-and-paper oriented) simplification of the run-time stack. The only information used from the runtime stack are the parameter values, the return values, and the calling order.

The trace can be very effective, and I've frequently used it in conjunction with other ways of explaining recursion. It can be confusing for a student to build a trace of a procedure which has multiple calls to itself. The Fibonnaci series is an easy example of such multiple calls, Ackermann's function (given later), is notoriously more difficult.

4. The recursion tree

An effective illustration of recursion is the *recursion tree*, (good examples are in [KLT91], [NyL92], [Sed88]). The recursion tree is a tree where each node is the "current environment." That is, each node contains parameters and local variables. Using this technique, it is easy to identify a node as a particular procedure executing in a particular environment. The parent of a node is the procedure which called the node. The children of a node are the procedures which that node calls.

Because of the 2-D presentation, the recursion tree is easier to examine and use than the trace. It also has the great advantage that it is but a step to go from recursion tree to induction. To get to induction from the runtime stack or from the trace is much more difficult.

The recursion tree has a few disadvantages that are easily remedied in the activation tree. First, the nodes of the recursion tree do not record the returning value of the node. Second, confusion can arise when building a recursion tree for a procedure which has more than two calls to itself. Because one is tracing through code while building the recursion tree, it is important to keep track of which procedure call *in the code*, one is returning from. This can be especially confusing when the recursive calls are embedded in different branches of a conditional. Again, Ackermann's function is a classic example of this difficulty.

5. The activation tree

The activation tree is a combination of the run-time stack and the recursion tree. The stack frames of the runtime stack contain additional required information, specifically the return value and the return address, the topological organization of the recursion tree makes it easier to follow the dynamic execution of the program. It has the advantages of both recursion tree and runtime stack. It does not have the disadvantages of either. It is a simple extension to the recursion tree, but has been surprisingly effective among my students.

3. The Activation Tree

To explain the activation tree, I'll just give the cookbook instructions I give to my students. I'll use Ackermann's function to illustrate.

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

INSTRUCTIONS:

1. Build a template of the activation record instance. It must include function name, parameters, and return value.

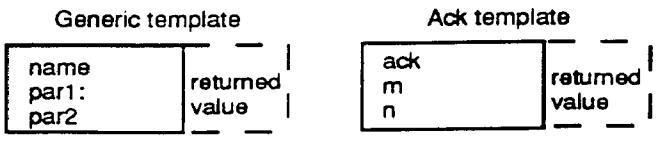

Figure 1 Activation Record Instance

2. Examine the code. If the procedure calls itself more than once, label each call to itself with a unique number. Note the numbers 1, 2, and 3 which subscript the three calls to ack. (Aside, this label is an abstraction of the return address).

```
int ack(int m, int n) {
    if (m==0) return (n + 1);
    if (n==0) return (ack1 (m-1,1) );
    return (ack2 (m-1, ack3 (m, n-1) ) );
    }
```

3. Draw the activation tree by tracing through the program. Begin with the first invocation of the procedure and draw, as the root of the activation tree, the template with parameters and local variables filled in.

4. Any call to any procedure is recorded in the activation tree as a child node of the current node. To execute the called procedure, move down to that new child node.

(a) If the current node makes several procedure calls, the children nodes are drawn left to right in the order they are called. See Figure 2.

(b) If arguments are calculated by making a recursive call, those arguments must be evaluated before they can be passed to any function. Make a normal (recursive) procedure call, recording it in the activation tree as a child of the current node.

(c) If an argument is calculated through a recursive procedure call, then passed to a (different) recursive call (as in Ackermann's function, ack_3), first evaluate the arguments, then call the procedure with the argument values. That is, there will be one child for each recursive argument evaluation, then one child for the function using those arguments.

5. A return from a procedure means filling in the returned value and moving back up the tree to the parent. Leaves of the tree correspond to procedures which do not call other procedures.

6 The dynamic execution of the program follows depth first traversal of the activation record tree.

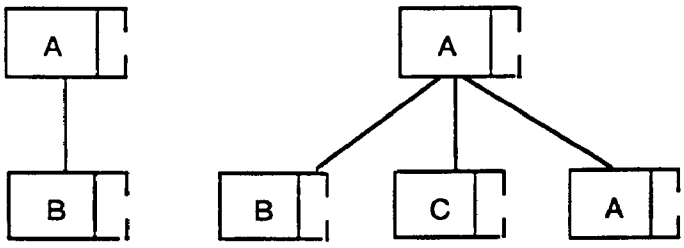

Figure 2
1st tree: A invokes B
2nd tree: A invokes B, C, A, in order

To complete the example, I give first the trace of Ackermann's function when invoked with parameter list (2,1); then its activation tree is given in Figure 3. Certain invocations and returns on the trace are labelled, which correspond to labels on the corresponding activation tree. This labelling is for illustrative purposes only.

```
> (ack 2 1)
Entering: ACK, Argument list: (2 1)        //(a)
 Entering: ACK, Argument list: (2 0)
  Entering: ACK, Argument list: (1 1)      //(b)
   Entering: ACK, Argument list: (1 0)
    Entering: ACK, Argument list: (0 1)    //(c)
    Exiting: ACK, Value: 2
   Exiting: ACK, Value: 2
   Entering: ACK, Argument list: (0 2)
   Exiting: ACK, Value: 3
  Exiting: ACK, Value: 3
 Exiting: ACK, Value: 3
 Entering: ACK, Argument list: (1 3)
  Entering: ACK, Argument list: (1 2)      //(d)
   Entering: ACK, Argument list: (1 1)
    Entering: ACK, Argument list: (1 0)
     Entering: ACK, Argument list: (0 1)
     Exiting: ACK, Value: 2
    Exiting: ACK, Value: 2
    Entering: ACK, Argument list: (0 2)
    Exiting: ACK, Value: 3
   Exiting: ACK, Value: 3
   Entering: ACK, Argument list: (0 3)
   Exiting: ACK, Value: 4
  Exiting: ACK, Value: 4
  Entering: ACK, Argument list: (0 4)      //(e)
  Exiting: ACK, Value: 5
 Exiting: ACK, Value: 5
Exiting: ACK, Value: 5
```

A second recursion example is the calculation of the Fibonacci number. The inductive definition is given in section 2 of this paper. A trace of a call to Fib(5) follows. Figure 4 shows the activation tree for Fib(5).

```
Entering: FIB , Argument list: (5)
  Entering: FIB , Argument list: (4)
    Entering: FIB , Argument list: (3)
      Entering: FIB , Argument list: (2)
      Exiting: FIB , Value: 1
      Entering: FIB , Argument list: (1)
      Exiting: FIB , Value: 1
    Exiting: FIB , Value: 2
    Entering: FIB , Argument list: (2)
    Exiting: FIB , Value: 1
  Exiting: FIB , Value: 3
  Entering: FIB , Argument list: (3)
    Entering: FIB , Argument list: (2)
    Exiting: FIB , Value: 1
    Entering: FIB , Argument list: (1)
    Exiting: FIB , Value: 1
  Exiting: FIB , Value: 2
Exiting: FIB , Value: 5
```

4. Moving to Induction

Once a student has control of the necessary detail of recursion, it is then important to begin building a more expert level chunking ability. The student needs to be able to move from the gritty detail to the more abstract, inductive definition of the function being performed through recursion.

Among the methods described above, the activation tree is a convenient method for making this step. The topology of the tree is derived from the execution, and the execution is closely tied to the inductive definition (which was used to write the function). For example, the activation tree in Figure 4 shows that the node for Fib(5) has two children: Fib(4) and Fib(3). That is, the value calculated for Fib(5) must use the values calculated for Fib(4) and Fib(3). The activation tree has the return values placed in the right-side box, making the return value accessible to the student. According to the definition of Fib, the values for Fib(4) and Fib(3) must be added together in order to obtain the value for Fib(5). If we replace the "5" in Fib(5) by "n", then the activation tree tells us that Fib(n) is a function of Fib(n-1) and Fib(n-2).

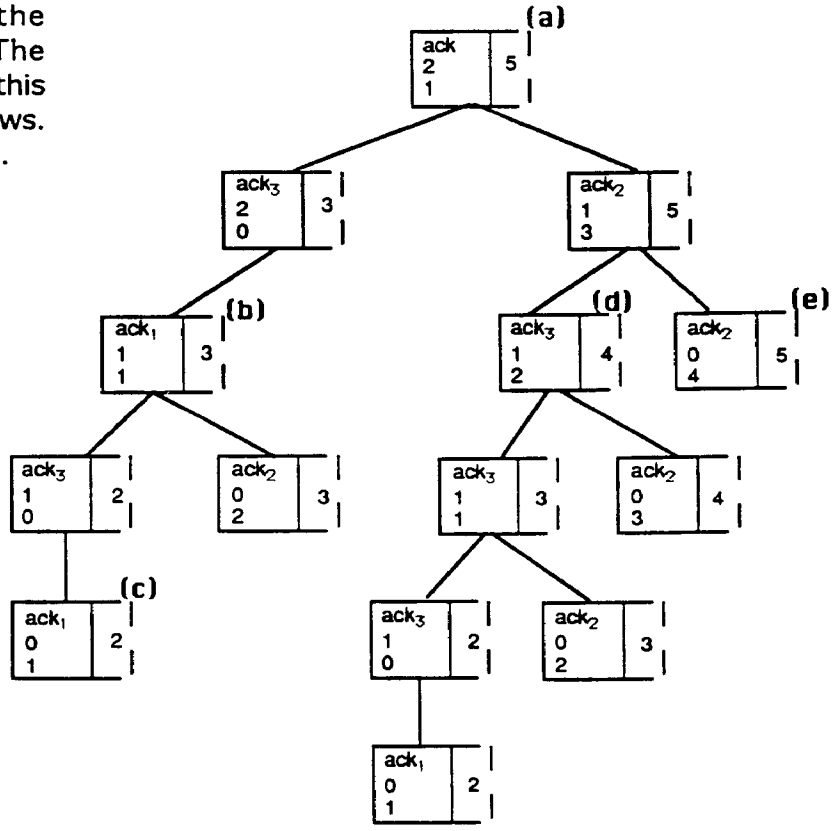

Figure 3 Activation tree for Ackermann(2,1)

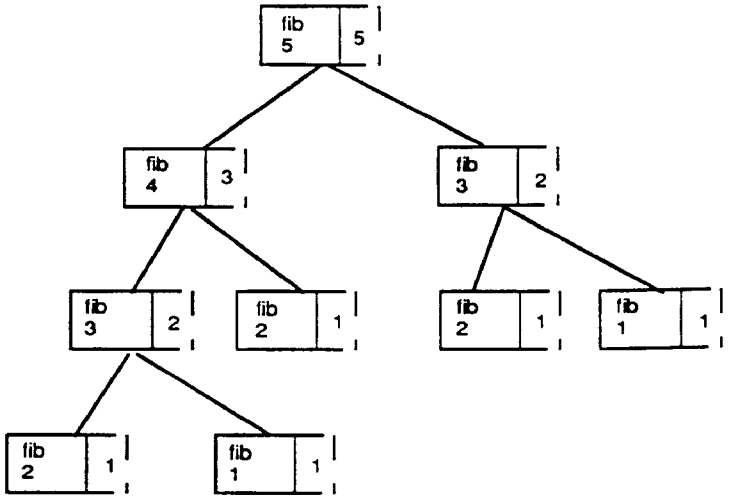

Figure 4 Activation tree for Fibonacci(5)

The leaves of the activation tree correspond to the base case(s) of the inductive definition. Thus, Fib(2) returns, immediately, with the value of 1.

******Recursion Continued On Page 14******

Decker R. and Hirshfield, S.: "The top 10 reasons why Object-Oriented programming can't be taught in CS1". *ACM SIGCSE Bulletin*, **26**(1), 51-55, 1994.

Mody, R.P.: "C in education and software engineering". *ACM SIGCSE Bulletin*, **23**(3), 45-56, 1991.

Pyott, S. and Sanders, I.: "ALEX: an aid to teaching algorithms". *ACM SIGCSE Bulletin*, **23**(3), 36-44, 1991.

Sakkinen, M.: "The darker side of C++ revisited". *Structured Programming*, **13**(4), 155-178, 1992.

Solway, E.: "Should we teach students to program". *Comm ACM*, **36**(10), 21, 1993.

Terry, P.D.: "Umbriel - a minimal programming language". Submitted to *ACM SIGPlan Notices*, 1995.

Wirth, N.: "Pascal-S: a subset and its implementation", in *Pascal - The Language and its Implementation*, John Wiley, Chichester, 1981.

Wirth, N,: "The programming language Oberon". *Software - Practice and Experience*, **18**, 671-690, 1988.

Wirth, N.: *Programming in Modula-2 (3rd edition)*. Springer-Verlag, Berlin, 1985.

**********Recursion Continued From Page 6**********

Because the inductive definition is a compact and very general description of recursion, it is important to understand it. The activation tree, with its visually oriented structure, offers a convenient stepping stone to thinking inductively.

5. Conclusion

While recursion can be subtle, the recursive problems given to beginning programmers, Fibonnaci series, factorial function, Towers of Hanoi, etc., are not particularly difficult. Yet explaining even simple examples of recursion to beginning programmers is difficult. This is not because of subtlety, but because of the complexity of tracing through code where there are many "pending" procedures with the same name.

There are several common approaches for helping students understand recursion: the inductive definition, the runtime stack, the trace and the recursion tree. To these approaches, this paper adds the activation tree. In topology, the activation tree is the recursion tree. In environment information, it is the runtime stack. Students have found the activation tree, especially in combination with the trace, to make recursion clear.

References

[Ast94] O. Astrachan, Self-reference is an illustrative essential, *SIGCSE Conference Technical Proceedings*, ACM, Phoenix, Arizona, March 1994, pp 238-242.

[KLT91] R.L. Kruse, B.P. Leung, C.L. Tondo, *Data Structures and Program Design in C*, Prentice Hall: New Jersey, 1991.

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

[NyL92] L. Nyhoff, S. Leestma, *Data Structures and Program Design in Pascal, 2nd Ed.*, Macmillan:New Yrok, 1992.

[Sed88] R. Sedgewick, *Algorithms, 2nd Ed.*, Addison-Wesley: Reading, Massachusetts, 1988.