PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

WHY RECURSION?

Jeffrey S. Rohl

Department of Computer Science
University of Western Australia

ABSTRACT

Recursion as a programming technique has been with us for over two decades now, and yet it still retains a certain mystery. In this paper we consider the objections to it and the claims for it.

1. INTRODUCTION

Many programming texts use Euclid's algorithm for calculating the highest common factor (*HCF*) of two integers *p* and *q* as one of their simple examples. The description of the algorithm usually goes something like this: "To find the HCF first divide *p* by *q* and calculate the remainder, *r*. If *r* = 0 then *q* is the HCF; otherwise repeat the process with *q* and *r* taking the place of *p* and *q*." From this description an iterative solution along the lines of Fig. 1 is usually presented (though if the example comes early it is expressed as a program rather than a procedure of course).

```
function HCF(p,q : integer) : integer;
    var r : integer;
    begin
    r := p mod q;
    while r <> 0 do
        begin
        p := q;
        q := r;
        r := p mod q
        end;
    HCF := q
    end
```

Fig. 1 A nonrecursive version of the HCF procedure

Yet the description given almost begs for the recursive procedure such as that of Fig. 2.

```
function HCF(p,q : integer) : integer;
    var r : integer;
    begin
    r := p mod q;
    if r = 0 then HCF := q else HCF := HCF(q,r)
    end
```

Fig. 2 A recursive version of the HCF procedure

Why then does the first solution seem more natural to writers and teachers? The simple (simplistic?) answer is that most writers and teachers either learned to program in the fifties or early sixties when recursion was just beginning to appear, or were themselves taught by people reared in that period. There seems to be a collective feeling for iterative solutions as against recursive ones, though this feeling is certainly buttressed by cogent arguments.

2. THE OBJECTIONS

What are these arguments against recursion? There seem to be four, which we discuss in turn.

(a) It is expensive of space: This is quite a strong argument since each invocation of the recursive procedure requires an activation record consisting of links, parameters and local variables. Let us make the simplifying assumption that all these quantities require a word each and that there are two links. Then the recursive procedure for HCF gives $5n$ words, where n is the number of recursive invocations, whereas the non-recursive version requires a constant 3. Whether this is important depends very much on the value of n. It so happens that for this example n must be small: if F_i is the i^{th} Fibonacci number then it is bounded by $u-3$ where F_u is the largest Fibonacci number represented by a variable of type *integer*. Similar statements apply to other numerical procedures such as that for factorial.

If on the other hand, the procedure is processing the elements of a list or an array, then n is usually related to the number of elements in this list or array, and this might be quite large; and in a program manipulating a small number of large lists it could be quite crucial. The decision on whether or not to use recursion in this situation is quite a nice one especially where the non-recursive procedure requires a stack.

When we move onto more complex data structures such as a tree, the space required by activation records is less significant since n is generally related to the height of the tree which, for reasonably balanced trees, is a logarithmic function of the number of nodes. The same is true but even more so for more general trees, including the search trees of combinatorial problems.

(b) It is expensive of time: This objection has in general lost much of its validity. If a compiler writer implements procedure calls, not by a short sequence of open code but by a call to a subroutine, then calling procedures is expensive, and the complaints about the time penalty of recursion are based on experience with these compilers.

If we consider any procedure written both recursively and non-recursively we find that, in general, the same operations take place and that in general they take place in the same order. The difference lies in the control structures: a recursive call or a traverse of a loop. Although the procedure call is the more expensive, the significance of this becomes correspondingly less as the body of the procedure becomes more complex. The HCF example is probably the most unsympathetic from the recursive point of view since the body is very small. Fig. 3 gives an analysis, in terms of the number of iterations/recursions n, of the operations involved.

	Weight	Non-recursive (Fig.1)	Recursive (Fig.2)
Assignments	1	$3n + 2$	$2n + 2$
Mod	4	$n + 1$	$n + 1$
Comparisons	1	$n + 1$	$n + 1$
Procedure calls	5	1	$n + 1$
Parameters passed	1	2	$2n + 2$
Weighted figure		$8n + 14$	$14n + 14$

Fig. 3 An analysis of the HCF procedures

The weights used to produce the weighted average are rather arbitrary (and reflect the writer's feeling of what they should cost!) On the CYBER they are reasonably accurate. The times to calculate the HCF of F_{24} and F_{23}, so that $n = 21$, were 700 μsecs and 1120 μsecs respectively (to some rather variable accuracy). This then sets an upper limit on the time penalty of recursion.

In more complex cases where the non-recursive procedure has to maintain a stack, the balance changes and the speed of the algorithms becomes more nearly equal. Indeed there is evidence (Fike 1975), (Rohl 1976) that a recursive procedure can be the more efficient.

(c) I can't understand it: There are those who have no need for recursion and for them the whole of this discussion is simply irrelevant. There are many others, however, for whom recursion would be useful if only they could understand it. Their inability to understand is a severe problem, and an indictment of those of us whose job it is to teach them and have failed. It is perhaps significant that none of the introductory texts on Pascal (assuming here that Wirth [1976] and Alagic & Arbib [1978] are not introductory) give the subject more than a cursory treatment.

Once recursion is mastered, it is difficult to believe that some non-recursive procedures are easier to understand than their recursive equivalents. Consider a procedure for producing a copy of a list, assuming the definitions:

```
type listptr = ↑node;
     node = record
              item : itemtype;
              next : listptr
            end
```

where *itemtype* is left unspecified.

Fig. 4 gives a non-recursive version adapted from the function given by Alagic and Arbib.

```
procedure copy(ℓ : listptr; var ℓ1 : listptr);
     var  p, pred : listptr;
     begin
     if   ℓ = nil then ℓ1 := nil
     else
          begin
          new(ℓ1);
          ℓ1↑.item := ℓ↑.item;
          pred := ℓ1; ℓ := ℓ↑.next;
          while ℓ <> nil do
               begin
               new(p);
               pred↑.next := p;
               p↑.item := ℓ↑.item;
               pred := p; ℓ := ℓ↑.next
               end;
          pred↑.next := nil
          end
     end
```

Fig. 4 A non-recursive procedure for copying a list

Is this procedure really easier to understand than the recursive one given in Fig. 5?

```
procedure copy(ℓ : listptr; var ℓ1 : listptr);
     begin
     if   ℓ = nil then ℓ1 := nil
     else
          begin
          new(ℓ1);
          ℓ1↑.item := ℓ↑.item;
          copy(ℓ↑.next, ℓ1↑.next)
          end
     end
```

Fig. 5 A recursive procedure for copying a list

For those uninitiated in recursion it may be, so that it seems that the solution is an educational one. We must see to it that the mode of thought involved in recursion is explained and that significant procedures are written using it.

(d) The language I use doesn't allow it: It is certainly true that Fortran forbids recursion and that most assembly languages give no help in its implementation. However, the problem of mechanistically converting recursive procedures to non-recursive ones has received a lot of attention. (See Griffiths [1975] for linear recursion, Knuth [1974] and Bird [1977] for binary recursion, and Rohl [1977] for recursion in combinatorial problems.) Thus it is possible to regard recursion as a design tool even where it may not be available as an implementation tool.

3. THE CLAIMS

The discussion so far has only been a partial answer to the four objections. We consider now four advantages.

(a) In appropriate situations it more naturally matches the problem: we have already given the example of a list copying procedure. Since the recursive version of that procedure is vulnerable to the space argument, we give another example: that of adding an element to a binary search tree. Fig. 6 gives a recursive version assuming the definitions:

```
type treeptr = ↑node;
     itemtype = record
                  key : keytype;
                  info : infotype;
                end;
     node = record
              left : treeptr;
              item : itemtype;
              right : treeptr
            end;
```

where *infotype* is left unspecified.

```
procedure insert(newitem : itemtype; var t : treeptr);
     begin
     if   t = nil then
          begin
          new(t);
          with t↑ do
               begin
               item := newitem;
               left := nil; right = nil
               end
          end
     else with t↑ do
          if   newitem.key = item.key
               then writeln('item already on tree')
          else if newitem.key < item.key
               then insert(newitem, left)
          else {if newitem.key > item.key then}
               insert(newitem, right)
     end
```

Fig. 6 A recursive procedure for inserting an element in a tree

The recursive procedure enables us to avoid the *trailing pointer problem* and Barron's *protasis problem*, as a comparison with Fig. 7 graphically illustrates.

```
procedure insert(newitem : itemtype; var t : treeptr);
     var  t1, t2 : treeptr;
          branch : (l, r);
          found : Boolean;
     begin
     new(t2); t2↑.right := t;
     t1 := t; t := t2; branch := r;
     found := false;
     while (t1 <> nil) and not found do
          with t1↑ do
               begin
               t2 := t1;
               if   newitem.key = item.key then
                    begin
                    writeln('item already on tree');
                    found := true
                    end
               else if newitem.key < item.key then
                    begin
                    t1 := t1↑.left; branch := l
                    end
               else {if newitem.key > item.key then}
                    begin
                    t1 := t1↑.right; branch := r;
                    end
               end;
     if   t1 = nil then
          begin
          new(t1);
          with t1↑ do
               begin
               item := newitem;
               left := nil; right := nil
               end
          if   branch = l then t2↑.left := t1
               else t2↑.right := t1;
          end;
     t := t↑.right
     end
```

Fig. 7 A non-recursive procedure for inserting an element in a tree

(b) In many situations such procedures are easier to prove: For a linear recursive procedure, its proof is almost trivial, since the structure of the procedure mirrors directly the mathematical formulation. The proof process is essentially that of the induction used in the proof of the underlying mathematics. Perhaps we are saying that it is the proof of the mathematics rather than the proof of the program that is important.

We give now a more difficult procedure, one for generating permutations in pseudo-lexicographical order, which we shall also use in later sections. We call the procedure *everyman* because every man and his brother seem to have discovered it. It assumes the definitions:

```
type mark = {any enumeration or subrange};
     range = 1 .. max;
     marksarray = array[range] of mark;
```

where max is the cardinality of *mark*, and uses :=: as an interchange operator.

```
procedure everyman(m : marksarray; n : range);

     procedure perm(k : range);
          var  i : range;
          begin
          for  i := k to n do
               begin
               m[k] :=: m[i];
               if k = n-1 then {process}
               else perm(k+1);
               m[k] :=: m[i]
               end
          end;

     begin
     perm(1)
     end
```

Fig. 8 The *everyman* procedure for generating permutations

The proof is simple. Suppose that a call $perm(k+1)$

(i) Leaves the marks in $m_1 \rightarrow m_k$ untouched;

(ii) Ensures that all permutations of the marks in $m_{k+1} \rightarrow m_n$ are produced in turn;

(iii) Returns $m_{k+1} \rightarrow m_n$ to its original state.

This is trivially true when $k+1 = n-1$.

Then a call $perm(k)$:

(i) Leaves the marks in $m_1 \rightarrow m_{k-1}$ untouched since the procedure does not reference them;

(ii) Ensures that all the permutations of the marks $m_k \rightarrow m_n$ are produced in turn because all possible choices for m_k (available in $m_k \rightarrow m_n$) are chosen and $perm(k+1)$ called after each choice;

(iii) Returns $m_k \rightarrow m_n$ to its original state.

Since *everyman* calls $perm(1)$ it follows that all permutations of the marks in m are produced.

The proof is not always so easy, of course. We leave the reader to prove a related algorithm due to Heap [1963] given in Fig. 9.

```
procedure Heap(m : marksarray; n : range);

     procedure perm(k : range);
          var  i, p : range;
          begin
          if   k = n-1 then {process}
          else perm(k+1);
          for  i := k + 1 to n do
               begin
               if   odd(n-k) then p := i else p := n;
               m[p] :=: m[k];
               if   k = n-1 then {process}
               else perm (k+1)
               end
          end;

     begin
     perm(1)
     end
```

Fig. 9 Heap's algorithm for generating permutations

A non-recursive version of *everyman* is given in Fig. 10 and the reader is encouraged to prove it directly.

```
procedure everyman(m : marksarray; n : range);
     var  i : array[range] of range;
          k : range;
          complete, downagain : Boolean;

     begin
     k := 1;
     i[1] := k;
     complete := false;
     repeat
          m[k] :=: m[i[k]];
          while k <> n-1 do
               begin
               k := k + 1;
               i[k] := k;
               m[k] :=: m[i[k]]
               end;
          process;
          downagain := false;
          repeat
          m[k] :=: m[i[k]];
               if   i[k] <> n then
                    begin
                    i[k] := i[k] + 1;
                    downagain := true
                    end
               else
                    if k = 1 then complete := true
                    else k := k-1
          until downagain or complete
     until complete
     end
```

Fig. 10 A non-recursive version of *everyman*

(c) In many situations such procedures are easy to analyse: We illustrate this by reference to the *everyman* procedure again. Let us ignore for the moment the details of what we choose to measure, and assume that:

- a is the count inside the loop at level $n-1$
- b is the count outside the loop at level $n-1$
- c is the count inside the loop at the other levels,
- d is the count outside the loop at the other levels.

If T_k is the count for a complete activation at level k then we have:

$$\begin{aligned} T_k &= (n-k+1)*(T_{k+1}+c) + d \quad , k \neq n-1 \\ &= 2 \times a + b \quad , k = n-1 \end{aligned}$$

From this we can calculate T_1 as:

$$\begin{aligned} T_1 = n \times \; & [a + \\ & b \times \tfrac{1}{2!} + \\ & c \times (\tfrac{1}{2!} + \tfrac{1}{3!} + \ldots) + \\ & d \times (\tfrac{1}{3!} + \tfrac{1}{4!} + \ldots)] \\ = n \times \; & [a + \\ & (b+c) \times \tfrac{1}{2!} + \\ & (c+d) \times \tfrac{1}{3!} + \\ & (c+d) \quad \tfrac{1}{4!} + \\ & \vdots] \end{aligned}$$

Fig. 11 gives an analysis of *everyman* with respect to some higher-level constructs.

		Parameters				Terms				
	Weight	a	b	c	d	a	$\frac{b+c}{2!}$	$\frac{c+d}{3!}$	$\frac{c+d}{4!}$	Total
Assignments	1	6	0	6	0	6	3	1	$\frac{1}{4}$	$10\frac{1}{4}n!$
Arithmetic	1	1	0	2	0	1	1	$\frac{1}{3}$	$\frac{1}{12}$	$2\frac{5}{12}n!$
Subscripts	1	8	0	8	0	8	4	$1\frac{1}{3}$	$\frac{1}{3}$	$13\frac{2}{3}n!$
Comparisons	1	1	0	1	0	1	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$	$1\frac{17}{24}n!$
Loop entries	1	0	1	0	1	0	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$	$\frac{17}{24}n!$
Loop traverses	3	1	0	1	0	1	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$	$1\frac{17}{24}n!$
Parameters	1	0	0	1	0	0	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$	$\frac{17}{24}n!$
Calls	5	0	0	1	0	0	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$	$\frac{17}{24}n!$
Weighted		19	1	26	1	19	$13\frac{1}{2}$	$4\frac{1}{2}$	$1\frac{1}{8}$	$38\frac{1}{8}n!$

Fig. 11 An analysis of *everyman*

From a detailed analysis such as this we can determine the effects of proposed transformations on a procedure to improve its performance. With *everyman*, for example, we could consider, among others, the following possibilities:

(i) During all interchanges at one level the same element (the initial m_k) takes place in all interchanges, and reinterchanges. We could save on both assignments and subscriptings by storing this value locally outside the loop so that the interchange within the loop required only two assignments instead of three. Further we could avoid restoring m_k in the interchange sequence since on the next traverse it would be immediately overwritten. That is, we could replace the loop of *everyman* by:

```
temp := m[k];
for  i := k to n do
     begin
     m[k] := m[i];   m[i] := temp;
     if   k = n-1 then {process}
     else perm(k+1);
     m[i] := m[k]
     end;
m[k] := temp
```

(ii) The interchange and reinterchange that takes place on the first traverse of each loop is redundant since it simply interchanges m_k with itself. We could recognise this by dealing with it outside the loop and reducing the number of traverses by one. Note that this means that a new derivation must take place. The new result is:

$$
\begin{aligned}
T_1 \quad = n! \times [(a+b+c) \times \tfrac{1}{2!} \\
+ \; d \times \tfrac{1}{3!} \\
+ \; d \times \tfrac{1}{4!} \\
\vdots \quad]
\end{aligned}
$$

Note, too, that the processing must now take place at two different places in the text, which itself may imply some cost.

(iii) The test for determining when the recursion is to terminate is constant within the loop. It may be taken outside by splitting the loop into two and using the test to determine which loop is to be obeyed. Further the loop at the bottom level is obeyed only once and can be replaced by its body.

(iv) We could stop the recursion one level later as in Wirth. This involves a third analysis which we leave to the reader.

Fig. 12 gives the results of the analysis of the above suggestions together with times in msecs of running them on a CYBER 73 for $n = 6$.

	Weighted Parameters					
	a	b	c	d	Total	Time
Basic procedure	19	1	26	1	$38\frac{1}{8}$n!	130
Mod(i)	13	5	20	5	$30\frac{17}{24}$n!	108
Mod(i) → (ii)	13	14	14	15	$23\frac{5}{8}$n!	77
Mod(i) → (iii)	-	16	18	15	$20\frac{1}{8}$n!	67
Mod(i) → (iv)	-	1	18	14	$28\frac{11}{12}$n!	101

Fig. 12 An analysis of improvements to *everyman*

Furthermore, similar analyses (or the same ones stopping earlier) enable us to determine whether the same techniques are more or less efficacious if we want the permutations to be r at a time rather than n at a time. This is relevant to adaptations of the procedure for, say, topological sorting or other procedures where inspection of the first r elements of a permutation may enable all $(n-r+1)!$ permutations starting with those r elements to be removed from consideration without being generated.

(d) They are adaptable: This is rather a difficult claim to justify yet it is interesting to note how often workers express their amazement that minor changes to a program can produce a highly desirable variant. Here we will simply illustrate by means of the classical *n-queens problem*: that is, the problem of determining how n queens may be placed on an $n \times n$ chessboard so that no queen is under attack from any other. If we represent the solution as an array m where m_i gives the column in which the queen on row i is placed, then since there can only be one queen in each row and one queen in each column, it follows that m must be a permutation of the integers 1 to n.

Thus a permutation generation procedure can be adapted to solve the n-queens problem by testing each permutation to see whether it corresponds to a board in which no queen is under threat along the diagonals. Further we can test partial permutations as they are generated to see whether the queen, represented by the latest element to be added to the permutation, is under attack since, if it is, there is no point building on the partial permutation. Fig. 13 gives a procedure based on *everyman* which uses the traditional technique for testing the diagonals.

```
procedure queens(n : range);
    const max1 = {the value of max - 1};
          max2 = {the value of 2 × max};
    type  mark = 1 .. max;
    var    m : array[range]of mark;
         upl : array[-max1 .. max1] of Boolean;
         upr : array[2 .. max] of Boolean;
           i : integer;

procedure perm(k : range);
    var  i, mi : range;
         temp : mark;
    begin
    temp := m[k];
    for  i := k to n do
         begin
         mi := m[i];
         if upl[k-mi] and upr[k+mi] := then
              begin
              upl[k-mi] := false; upr[k+mi] := false;
              m[k] := mi; m[i] := temp;
              if k = n-1 then
                   begin
                   if   upl[n-m[n]] and
                        upr[n+m]n]] then process
                   end
              else perm(k+1);
              m[i] := mi;
              upl[k-mi] := true; upr[k+mi] := true
              end
         end;
    m[k] := temp
    end;

begin
for  i := 1 to n do m[i] := i;
for  i := 1-n to n-1 do upl[i] := true;
for  i := 2 to 2 × n do upr[i] := true;
perm(1)
end
```

Fig. 13 The n-queens problem

4. CONCLUSIONS

The reader will have noticed that the claims for recursion have generally been prefaced by the phrase "in many situations". The paper does not claim that recursion should be used for everything (though the author still nurtures the dream of teaching an introductory programming course this way). We simply want to say that recursion is a very powerful tool on the appropriate occasion and that it should not be dismissed as too esoteric for practical use.

5. ACKNOWLEDGEMENT

I should like to thank M.S. Palm who tested, timed and instrumented all the procedures given here.

REFERENCES

1. ALAGIC, S and ARBIB, M.A. (1978): "The Design of Well-Structured and Correct Programs", Springer-Verlag.

2. BIRD, R.S. (1977): "Notes on Recursion Elimination", Comm ACM, Vol.20, p.434.

3. FIKE, C.T. (1975): "A Permutation Generation Method", Computer Journal, Vol. 18, p.21.

4. GRIFFITHS, M. (1975): "Requirements for and Problems with Intermediate Languages for Programming Language Implementation" (Lecture notes for the NATO International Summer School, Marktoberdorf, W. Germany).

5. HEAP, B.R. (1963): "Permutations by Interchanges", Computer Journal, Vol. 6, pp 293-4.

6. KNUTH, D.E. (1974): "Structured Programming with Goto Statements", Computing Surveys, Vol. 6, p.261.

7. ROHL, J.S. (1976): "Programming Improvements to Fike's Algorithm for Generating Permutations", Computer Journal, Vol 19, p. 156.

8. ROHL, J.S. (1977): "Converting a Class of Recursive Procedures into Non-recursive Ones", Software - Practice & Experience, Vol. 7, p.231.

9. WIRTH, N. (1976): "Algorithms + Data Structures = Programs", Prentice-Hall.