Towers of Hanoi: The Derivation of Some Iterative Versions

J. S. ROHL
Department of Computer Science, University of Western Australia, Nedlands 6009, Western Australia

This paper responds to a challenge by Hayes to convert the recursive solution of the Towers of Hanoi problem to the iterative form given in his paper. In the process it derives a number of other iterative solutions, including the 'binary digits' solution of Gardner and MacCallum.

Received September 1985

1. INTRODUCTION

The Towers of Hanoi, a nineteenth-century toy, has been completely solved by the world's puzzlists, but over the last twenty years it has attracted the attention of the computer scientists, whose interest lies in the programming of the solution.

There are two reasons for this interest. First, the problem has an elegant recursive solution, which is an excellent paradigm of binary recursion, being more interesting than, for example, the generation of Grey codes. For a long time, no iterative version was given in the computing literature (although Martin Gardner[1] described a couple of algorithms in his books), and the impression gained ground that an iterative solution was somehow rather difficult. This leads to the second reason for the interest in the problem: iterative procedures, in the style of Gardner, have appeared, which are also elegant. Furthermore, there appears to be little relation between the versions, and some authors, for example Hayes,[2] have thrown down the challenge:

It would be a very nontrivial exercise to convert (the recursive version) to (the non-recursive version), let alone convert it mechanically. In fact...I hereby offer it as a challenge to optimistic optimisers, and to those who make it their business to prove that equivalent programs are equivalent.

So far this has not been answered. In this paper we show how the two archetypal iterative procedures may be derived by transformation from a recursive version. In the process, a succession of iterative procedures will be derived, and a number of properties of the problem will be revealed.

2. DESCRIPTION

The problem has been described many times: almost every author has felt it necessary to describe it in his own words. We give here what we believe to be the original description in English given by Rouse Ball in 1892.[3]

It consists of three pegs fastened to a stand, and of eight circular discs of wood or cardboard, each of which has a hole in the middle through which a peg can be passed. These discs are of different radii, and initially they are all placed on one peg, so that the biggest is at the bottom, and the radii of the successive discs decrease as we ascend: thus the smallest disc is at the top. This arrangement is called the *Tower*. The problem is to shift the discs from one peg to another in such a way that a disc shall never rest on a disc smaller than itself and finally to transfer the tower (i.e. all the discs in their proper order) from the peg on which they initially rested to one of the other pegs.

Note that the problem explicitly mentions eight discs, so that a procedure to solve the problem is inherently parameterless. However, the puzzle is said to be derived from the mythical Tower of Bramah, which has 64 discs, and is traditional for procedures to solve the problem for any number of discs. Accordingly, we introduce:

```
type disc = 0..64
```

Note also that the problem does not refer to discs or pegs by name. Different solutions may do so, of course, and will in general do it differently. Accordingly, we adopt a two-level solution in which the outer procedure has only one parameter (the number of discs) and a simple body which, apart perhaps from some initialisation, simply calls the inner procedure which does all the 'real' work.

3. THE STANDARD RECURSIVE VERSION

The standard recursive procedure is based on the observation, first made by Rouse Ball,

The method...is as follows. (i) If initially there are n discs on peg A, the first operation is to transfer gradually the top $n-1$ discs from peg A to peg B, leaving the peg C vacant...(ii) Next, move the bottom disc to peg C. (iii) Then, reversing the first process, transfer gradually the $n-1$ discs from B to C.

Note that he has named the pegs! Almost every author has followed suit by defining a type to specify the pegs, sometimes an enumerated type, more often the subrange:

```
type peg = 1..3
```

Rouse Ball also numbered the discs, though he didn't specify the numbering system in detail. We follow the usual convention that discs are numbered consecutively from 1 starting from the smallest, so that the bottom disc of a tower size n is always numbered n.

If we choose to make as the special case the moving of a tower of size 0, the procedure *Hanoi* follows directly:

```
procedure Hanoi (n: disc);
  type
    peg = 1..3;
  procedure MoveDisc (k: disc; p1, p2: peg);
  begin
    writeln('Move disc', k:1, 'from', p1: 1, 'to', p2:1)
  end;{of procedure 'MoveDisc'}
```

```
procedure H (k: disc; p1, p2, p3: peg);
    begin
      if k < > 0 then
        begin
          H(k - 1, p1, p3, p2);
          MoveDisc(k, p1, p2);
          H(k - 1, p3, p2, p1)
        end
    end; {of procedure 'H'}
  begin
    H(n, 1, 2, 3)
  end;{of procedure 'Hanoi'}
```

4. ELIMINATING THREE OF THE PARAMETERS

Let us transform this recursive version into a form more amenable to the recursion elimination which follows. The aim is to remove all but the first parameter of *H*.

First, we notice that *p3* is redundant, since $p1+p2+p3 = 6$, so we eliminate it.

Secondly, we eliminate *p2*, though this will take a couple of steps. As a first step, we refer to the second peg, not by its name, *p2*, but by its direction, *d*, clockwise or anticlockwise from *p1*. (It is easiest to think of the pegs as being a triangular arrangement, as was the case with the original form of the toy.)

H and *Movedisc* must be modified to calculate the destination of the move from *p1* (renamed *p* since there is only one left) and *d*. To move a tower of size *k* from peg *p* to its neighbour in direction *d*, we must first move the tower of size $k-1$ to its neighbour in the opposite direction; then move the bottom disc in direction *d*; and finally move the tower of size $k-1$ again in the opposite direction. To this end it is convenient to introduce functions, *Opp*, which yields the opposite of a direction, and *Neigh*, which yields the number of the peg which is *p*'s neighbour in a given direction. To improve readability, we will, in this description, use functions and procedures rather than in-line code. In practice, of course, we might use other criteria.

The complete procedure *Hanoi* is:

```
procedure Hanoi (n: disc);
  type
    peg = 1..3;
    direction = (clock, anti);
  function Opp (d: direction): direction;
  begin
    if d = clock then
      Opp: = anti
    else
      Opp: = clock
  end;{of function 'Opp'}
  function Neigh (p: peg; d: direction): peg;
  begin
    Neigh: = (p+ord(d)) mod 3+1
  end;{of function 'Neigh'}
  procedure MoveDisc (k: disc; p: peg; d: direction);
  begin
    writeln('Move disc', k: 1, 'from', p: 1, 'to', Neigh
                                                (p, d):1)
  end;{of procedure 'MoveDisc'}
  procedure H(k: disc; p: peg; d: direction);
  begin
    if k < > 0 then
      begin
        H(k - 1, p, Opp(d));
        MoveDisc(k, p, d);
        H(k - 1, Neigh(p, Opp(d)), Opp(d))
      end
  end;{of procedure 'H'}
begin
  H(n, 1, clock)
end;{of procedure 'Hanoi'}
```

Notice that, at a recursive call the values of the variables *k* and *d* of the newly created activation record are related in a very simple way to the values of *k* and *d* in the calling activation: *k* is reduced by 1 and *d* is inverted. Thus *d* is redundant, since it defined by the parity of *k*, and so we can eliminate *Opp*, transform *H* and modify *MoveDisc* and *Neigh* accordingly.

```
procedure H (k: disc; p: peg);
begin
  if k < > 0 then
    begin
      H(k - 1, p);
      MoveDisc(k, p);
      H(k - 1, Neigh(p, odd(n - k + 1)))
    end
end;{of procedure 'H'}
procedure MoveDisc (k: disc; p: peg);
begin
  writeln('Move disc', k: 1, 'from', p: 1, 'to', Neigh
                                        (p, odd(n - k)): 1)
end;{of procedure 'MoveDisc'}
```

Note that in doing this transformation we have discovered a property of the solution to the Hanoi problem: a disc always moves in the same direction. Further, odd-numbered discs move in one direction, while even-numbered ones move in the opposite direction.

We now eliminate *p*. Any parameter called by value can be replaced by a global variable, provided there exists an inverse for each expression used as an actual parameter for it. Here the expressions and their inverses are:

p	*p*
$Neigh(odd(n-k+1))$	$Neigh(odd(n-k))$

To eliminate the parameter, we assign the global variable *p* its value before the call, and reassign it its original value by using the inverse after the call. Both *Neigh* and *MoveDisc* are modified to access *p* non-locally, too. If we eliminate $p: = p$ as redundant we produce for *H*:

```
procedure H (k: disc);
begin
  if k < > 0 then
    begin
      H(k - 1);
      MoveDisc(k);
      p: = Neigh(odd(n - k + 1));
      H(k - 1);
      p: = Neigh(odd(n - k));
    end
end;{of procedure 'H'}
```

The body of *Hanoi* itself is modified to assign *1* to *p*.

5. REMOVING THE POSTORDER STATEMENT

We now eliminate the postorder statement. Consider the following schema:

```
procedure C (x: xtype);
begin
  if P(x) then
    begin
      C(F1(x));
      S1(x);
      C(F2(x));
      S2(x)
    end
end
```

and suppose the the current invocation is the bottom one. Immediately after *S2*(x) is obeyed control returns to the previous invocation, and either *S1* or *S2* is obeyed, with the appropriate value of x, say x'. In the latter case, as soon as *S2*(x') is obeyed, control returns up one further invocation, and either *S1* or *S2* is obeyed with the appropriate value of x, say x''. And so on. Thus *S1* in any level is obeyed immediately after obeying a sequence of *S2* statements, one for each of the succeeding invocations in reverse order. Thus the *S2* statement can be replaced by a loop of *S2* statements placed immediately before the *S1* statement. In general this will require a stack, but if *F1* has an inverse, then the following schema:

```
procedure C (x: xtype);
begin
  if P(x) then
    begin
      C(F1(x));
      x1: = x;
      while P(x) do
        x1: = F1(x1);
      while x1 < > x do
        begin
        S2(x1);
        x1: = F1^{-1}(x1)
        end;
      S1(x);
      C(F2(x))
    end
end
```

together with a loop of *S2* statements after the initial call to *C*, is equivalent to the one above. In our example where the *Hanoi* procedure exits immediately after the call to *H*, we can forget this loop.

This second schema looks much worse than the original, as is usually the case with such transformations. However, for our particular example, both loops can be eliminated. With the appropriate substitutions, the first loop becomes:

```
k1: = k;
while k1 < > 0 do
  k1: = k1 - 1
```

and this simply assigns *0* to *k1*.

The other loop becomes:

```
while k1 < > k do
  begin
    p: = Neigh(odd(n - k1));
    k1: = k1 + 1
  end
```

Given that *k1* is initially 0, this corresponds to a sequence of statements:

```
p: = Neigh(odd(n));
p: = Neigh(odd(n - 1));
p: = Neigh(odd(n - 2));
          .
          .
p: = Neigh(odd(n - k + 1))
```

Consecutive pairs cancel, so that the total effect is null if k is odd, or just reduces to the last statement if k is even.

Thus we have, for *H*:

```
procedure H (k: disc);
begin
  if k < > 0 then
    begin
      H(k - 1);
      if odd(k - 1) then
        p: = Neigh(odd(n - k + 1));
      MoveDisc(k);
      p: = Neigh(odd(n - k + 1));
      H(k - 1);
    end
end; {of procedure 'H'}
```

Finally, we move the assignments of p into the *MoveDisc* procedure:

```
procedure MoveDisc (k: disc);
begin
  if odd(k - 1) then
    p: = Neigh(odd(n - k + 1));
  writeln('Move disc', k: 1, 'from', p: 1, 'to', Neigh
                                      (odd(n - k)): 1);
  p: = Neigh(odd(n - k + 1));
end;{of procedure 'MoveDisc'}
```

to produce for *H*:

```
procedure H(k: disc);
begin
if k < > 0 then
  begin
    H(k - 1);
    MoveDisc(k);
    H(k - 1);
  end
end;{of procedure 'H'}
```

Note that the form of *H* is identical to that of the initial version, a pure inorder procedure. Note, too, the clear separation of concerns: *MoveDisc* is concerned with determining pegs and writing out the moves, while *H* is almost completely control structure.

If we wished *Hanoi* to write out the moves in terms of discs and directions, for example:

Move disc 1 clockwise

we simply modify *MoveDisc* accordingly.

6. REPLACING THE RECURSION BY A STACK

The procedure H is in the form of an inorder binary recursive procedure. The elimination of recursion by utilising a stack has been well studied and a number of strategies have been described. We choose the technique due to Rohl[4] called substitution. Using that strategy for our inorder example, we get the following pair of schemata.

```
procedure C (x:xtype);                 procedure C (x:xtype);
                                         var
                                           s: stack of xtype;
                                           done: Boolean;
begin                                  begin
if P(x) then                             clear s;
  begin                                  repeat
    C(F1(x));                              while P(x) do
    S(x);                                    begin
    C(F2(x));                                push x onto s;
  end                                          x: = F1(x)
end;{of procedure 'C'}                       end;
                                         done: = s empty;
                                         if not done then
                                           begin
                                           pop x from s;
                                           S(x);
                                           x: = F2(x)
                                           end
                                         until done
                                       end;{of procedure 'C'}
```

Making the appropriate substitutions we produce:

```
procedure H (k: disc);
  var
    s: stack of disc;
    done: Boolean;
  begin
    clear s;
    repeat
      while k < > 0 do
        begin
          push k onto s;
          k: = k - 1
        end;
      done: = s empty;
      if not done then
        begin
          pop k from s;
          MoveDisc(k);
          k: = k - 1
        end
    until done
  end;{of procedure 'H'}
```

Since this procedure is no longer recursive, and is called only once, from within *Hanoi*, it could be unrolled: that is, the text could be substituted for the call, with the appropriate initialisation for the parameter. Again, for readability, we do not do so.

The inner loop of this procedure has its exit effectively in the middle, since on the last traverse half the body is skipped. This can be eliminated by using the transformation:

```
repeat              S1;
  S1;               while not P do
  if not P then       begin
    S2                  S2;
until P                 S1
                      end
```

which produces the procedure:

```
procedure H (k: disc);
  var
    s: stack of disc;
    done: Boolean;
begin
  clear s;
  while k < > 0 do
    begin
      push k onto s;
      k: = k - 1
    end;
  done: = s empty;
  while not done do
    begin
      pop k from s;
      MoveDisc(k);
      k: =k - 1;
      while k < > 0 do
        begin
          push k onto s;
          k: =k - 1
        end;
      done: = s empty;
    end
end;{of procedure 'H'}
```

This procedure can be tidied up. First, we eliminate the variable *done*, replacing its one applied occurrence by the value assigned to it, *s* **empty**. Secondly, we replace the loops on *k* by for-statements. Thus:

```
procedure H (k: disc);
  var
    s: stack of disc;
begin
  clear s;
  for k: = k downto 1 do
    push k onto s;
  while not(s empty) do
    begin
      pop k from s;
      MoveDisc(k);
      for k: = k - 1 downto 1 do
        push k onto s;
    end
end;{of procedure 'H'}
```

The procedures of this section are not in Pascal, which does not support stacks. It is a trivial matter, though, to implement these facilities using an array, and we leave it as an exercise for the interested reader.

7. REPLACING THE STACK BY A SET

The stack has an interesting property that we can capitalise on since:

- It is initialised to the value $\langle k, k-1, \ldots, 2, 1\rangle$ where the top of the stack is to the right.
- Whenever a value, say i, is popped we immediately push the values $\langle i-1, i-2, \ldots, 2, 1\rangle$.

That is, all the elements on the stack must be no greater than k, and each element is strictly less than the element beneath it. Thus the stack can be represented by a set of the integers $1..k$. To pop a value from the stack, we need a loop searching for the smallest element of the set; and to push $\langle k-1, k-2, ..., 2, 1\rangle$ we simply add $\{1..k-1\}$ to the set. This leads to the procedure:

```
procedure H (k: disc);
  var
    s: set of disc;
begin
  s: = [1..k];
  while s < > [] do
    begin
      k: = 1;
      while not (k in s) do
        k: = k+1;
      s: = s-[k];
      MoveDisc(k);
      s: = s+[1..k-1]
    end
end;{of procedure 'H'}
```

8. REPLACEMENT OF THE SET BY AN INTEGER

Now there is a one-to-one correspondence between subsets of the set $\{1..k\}$ and the integers $1..2^k-1$. It is the usual mapping that Pascal implementations adopt, except that the base set here starts at 1. Assuming the existence of an exponentiation function, *Power*, we have the equivalences:

s: **set of** *disc*	*s*: *natural*
s: = [1..*k*]	*s*: = *Power*(2, *k*) − 1
s < > []	*s* < > 0
not (*k* **in** *s*)	*s* **mod** *Power*(2, *k*) = 0
s − [*k*]	*s* − *Power*(2, *k* − 1)
s + [1..*k* − 1]	*s* + *Power*(2, *k* − 1) − 1

Thus we can transform *H*:

```
procedure H (k: disc);
  var
    s: natural;
begin
  s: = Power(2, k)-1;
  while s < > 0 do
    begin
      k: = 1;
      while s mod Power(2, k) = 0 do
        k: = k+1;
      s: = s-Power(2, k-1);
      MoveDisc(k);
      s: = s+Power(2, k-1)-1
    end
end;{of procedure 'H'}
```

We can now eliminate most of the powering. First, we combine $s := s - Power(2,k-1)$ and $s := s + Power(2, k-1)-1$ to get, simply, $s := s-1$. Secondly, we rephrase the inner while-statement to calculate the sequence of powers it uses. Finally, we express the outer loop as a for-statement, absorbing the $s := s-1$ statement:

```
procedure H (k: disc);
  var
    s, power2k: natural;
begin
  for s: = Power(2, k)-1 downto 1 do
    begin
      k: = 1;
      power2k: = 2;
      while s mod power2k = 0 do
        begin
          k: = k+1;
          power2k: = power2k*2
        end;
      MoveDisc(k);
    end
end;{of procedure 'H'}
```

From this transformation we deduce that the number of moves required is 2^n-1.

9. THE BINARY-DIGITS ALGORITHM

It is clear that the sequence which calculates *k* in the last version of *H* is simply determining the position of the least significant 1 in the binary representation of *s*, counting from 1. We will therefore introduce a function *LS1* to return this value. Now *s* ranges over the values 2^k-1 to 1. Since the position of the least significant 1 in *s* is the same as that of 2^k-s for all *s* in that range, the loop can be made to run forwards instead of backwards. Thus we arrive at the following procedure for *H*.

```
procedure H (k: disc);
  var
    s: natural;
begin
  for s: = 1 to Power(2, k)-1 do
    MoveDisc(LS1(s));
end;{of procedure 'H'}
```

This, probably the simplest iterative form known, is the algorithm described by Gardner. He did not give a formal description, however, and so did not specify the details of *MoveDisc*.

Let us now transform *H* by partially unrolling the loop on *s* by using the transformation:

```
for s: = 1 to 2*i+1 do     S(1);
  S(s)                     for s: = 1 to i do
                             begin
                               S(2*s);
                               S(2*s+1)
                             end
```

Now both 1 and $2*s+1$ are odd, and $2*s$ is even, so that:

$$LS1(1) = 1$$
$$LS1(2*s+1) = 1$$
$$LS1(2*s) = LS1(s)+1$$

Thus we arrive at the following procedure for *H*:

```
procedure H (k: disc);
  var
    s: natural;
begin
  MoveDisc(1);
  for s: = 1 to Power(2, k-1)-1 do
    begin
      MoveDisc(LS1(s)+1);
      MoveDisc(1);
    end
end;{of procedure 'H'}
```

This transformation reveals another property of the solution: moves of the smaller disc alternate with moves of the other discs.

Since half the moves involve the smallest disc, advantage can be obtained by introducing a procedure *MoveDisc1*, obtained by instantiation with $k = 1$ from *MoveDisc*. Both procedures are given below.

```
procedure MoveDisc (k: disc);
begin
  if odd(k-1) then
    p: = Neigh(odd(n-k+1));
  writeln('Move disc', k: 1, 'from', p: 1, 'to',
                              Neigh(odd(n-k)): 1);
  p: = Neigh(odd(n-k+1));
end;{of procedure 'MoveDisc'}
procedure MoveDisc1;
begin
  writeln('Move disc', 1: 1, 'from', p: 1, 'to',
                              Neigh(odd(n-1)): 1);
  p: = Neigh(odd(n));
end;{of procedure 'MoveDisc'}
```

Except for the initial movement of disc 1, calls for *MoveDisc* and *MoveDisc1* come in pairs. In each pair *p* is changed 3 times, so that it always specifies the source peg of the next move; and the destination peg is specified in relation to it. There are only 3 pegs, and therefore only 3 possible values for *p*. We ask therefore whether we can reduce the number of assignments to *p* to 1. We can do so because the 3 assignment statements:

```
if odd(k-1) then
  p: = Neigh(odd(n-k+1));
p: = Neigh(odd(n-k+1));
p: = Neigh(odd(n));
```

are equivalent to

```
p: = Neigh(odd(n-1))
```

as can be verified by trying all 4 cases of the parities of *k* and *n*. If we now assign to *p* only after the movement of the smallest disc, and express the source and destination pegs in relation to it, we arrive at the following procedures for *MoveDisc* and *MoveDisc1*.

```
procedure MoveDisc (k: disc);
begin
  writeln('Move disc', k: 1, 'from', Neigh(odd(n-k)):
    1, 'to', Neigh(odd(n-k+1)): 1);
end;{of procedure 'MoveDisc'}
procedure MoveDisc1;
begin
  writeln('Move disc', 1: 1, 'from', p: 1, 'to',
                              Neigh(odd(n-1)): 1);
  p: = Neigh(odd(n-1));
end;{of procedure 'MoveDisc1'}
```

I believe that this version is due to MacCallum.[2, 5]

Note that this transformation has revealed that when a larger disc is to be moved, the two pegs involved do not include the one on which the smallest disc is resting.

10. THE OPERATIONAL ALGORITHM

None of the procedures so far maintains a representation of the Towers, and all of them proceed by first determining the disc number, from which the pegs involved can be determined, generally with the aid of some other variable or variables. This means that the procedures are of little help to the person trying to solve the actual puzzle. This comment has been made often enough with respect to the initial recursive version, but it is equally valid for MacCallum's version. So now we introduce a representation for the Towers. It doesn't much matter what data structure we use, so we will follow Hayes and use a 3-element vector of lists, each list containing the numbers of the discs on that peg starting from the top. An appropriate definition is:

```
type
  listptr = ^node;
  node = record
    d: disc
    l: listptr
    end;
var
  tower: array [peg] of listptr
```

The initialisation is simple enough, and we do not give it. However, the *Move* procedures now must maintain the structure, as well as print out the move. Accordingly, it is convenient to merge them together again. Further, it will be convenient to determine the pegs involved within *H*, and transmit them as parameters, leaving *MoveDisc* to deal only with printing and data structure maintenance, thus:

```
procedure MoveDisc(k: disc; p1, p2: peg);
  var
    local: listptr;
begin
  writeln('Move disc', k: 1, 'from', p1: 1, 'to', p2: 1);
  local: = tower[p1]^.l;
  tower[p1]^.l: = tower[p2];
  tower[p2]: = tower[p1];
  tower[p1]: = local;
end;{of procedure 'MoveDisc'}
```

The procedure *H* follows directly from these changes:

```
procedure H (k: disc);
var
  s: natural;
begin
  MoveDisc(1, p, Neigh(odd(n-1)));
  p: = Neigh(odd(n-1));
  for s: = 1 to Power(2, k-1)-1 do
    begin
      k: = LS1(s)+1;
      MoveDisc(k, Neigh(odd(n-k)),
                         Neigh(odd(n-k+1)));
      MoveDisc(1, p, Neigh(odd(n-1)));
      p: = Neigh(odd(n-1))
    end
end;{of procedure 'H'}
```

Of course, this has not changed the algorithm: it has simply added the representation of the towers, which has, of course, introduced some redundancy. As it stands, H determines the value of *k* at each step, even though it is available in the data structure. Let us eliminate *k* entirely.

Consider the pair of statements:

```
k: = LS1(s)+1;
MoveDisc(k, Neigh(odd(n-k)), Neigh(odd(n-k+1)));
```

Given the definition of *Neigh*, the expression

$Neigh(odd(n-k))$

can be expressed as:

$(p + ord(odd(n-k)))$ **mod** $3+1$

which reduces to p **mod** $3+1$ if $n-k$ is even, and to $(p+1)$ **mod** $3+1$ if $n-k$ is odd. The expression *Neigh* $(odd(n-k+1))$ also reduces to one or other of these expressions.

If we knew k we could replace the call to *MoveDisc* by either:

MoveDisc(k, p **mod** $3+1$, $(p+1)$ **mod** $3+1$)

or:

MoveDisc(k, $(p+1)$ **mod** $3+1$, p **mod** $3+1$)

Of course, k is also at the head of the list associated with the destination peg. Therefore, since there are only two alternatives, we can search for the correct one:

```
p1: = p mod 3+1;
p2: = (p+1) mod 3+1;
if tower[p1]↑.d < tower[p2]↑.d then
  MoveDisc(tower[p1]↑.d, p1, p2)
else
  MoveDisc(tower[p2]↑.d, p2, p1)
```

which leads to the following procedure:

```
procedure H (k: disc);
  var
    s: natural;
begin
  MoveDisc(1, p, Neigh(odd(n-1)));
  p: = Neigh(odd(n-1));
  for s: = 1 to Power(2, k-1)-1 do
    begin
      p1: = p mod 3+1;
      p2: = (p+1) mod 3+1;
      if tower[p1]↑.d < tower[p2]↑.d then
        MoveDisc(tower[p1]↑.d, p1, p2)
      else
        MoveDisc(tower[p2]↑.d, p2, p1);
      MoveDisc(1, p, Neigh(odd(n-1)));
      p: = Neigh(odd(n-1))
    end
end;{of procedure 'H'}
```

To enable the comparison of the top elements of the lists to work when one of them is empty, we add to each list a node whose disc is larger than any real disc.

We still have the loop structure of H determined by algorithm rather than data structure. However, when the puzzle is solved, both *p1* and *p2* will have only the dummy disc on them, and we can use this information to terminate the loop. If we use a while-statement then *p1* and *p2* must be evaluated before the loop as well as inside it. The procedure H becomes:

```
procedure H (k: disc);
begin
  MoveDisc(1, p, Neigh(odd(n-1)));
  p: = Neigh(odd(n-1));
  p1: = p mod 3+1;
  p2: = (p+1) mod 3+1
  while tower[p1] < > tower[p2] do
    begin
      if tower[p1]↑.d < tower[p2]↑.d then
        MoveDisc(tower[p1]↑.d, p1, p2)
      else
        MoveDisc(tower[p2]↑.d, p2, p1);
      MoveDisc(1, p, Neigh(odd(n-1)));
      p: = Neigh(odd(n-1));
      p1: = p mod 3+1;
      p2: = (p+1) mod 3+1
    end
end;{of procedure 'H'}
```

This is almost the procedure given by Hayes. There are two differences. First, he used the property, noted in Section 4, that $p+p1+p2=6$, and calculated *p2* accordingly. Secondly, he used BCPL which allows exits from the middle of loops. The nearest we can do in Pascal is to use a repeat-statement, with a conditional-statement as its last statement:

```
procedure H (k: disc);
begin
  repeat
    MoveDisc(1, p, Neigh(odd(n-1)));
    p: = Neigh(odd(n-1));
    p1: = p mod 3+1;
    p2: = 6-p-p1;
    if tower[p1] < > tower[p2] then
      if tower[p1]↑.d < tower[p2]↑.d then
        MoveDisc(tower[p1]↑.d, p1, p2)
      else
        MoveDisc(tower[p2]↑.d, p2, p1);
  until tower[p1] = tower[p2]
end;{of procedure 'H'}
```

This is Hayes's algorithm.

11. CONCLUSIONS

This derivation has consisted of a long series of steps, some purely mechanical, some relying on a understanding of the structures being used. We have accomplished the first part of Hayes's challenge. It remains to be seen whether the second part, mechanical conversion, can be achieved.

REFERENCES

1. M. Gardner, *Mathematical Puzzles and Diversions*. Simon and Schuster, New York (1959).
2. P. J. Hayes, A note on the Towers of Hanoi problem, *The Computer Journal* **20** (3), 282–285 (1977).
3. W. W. Rouse Ball, *Mathematical Recreations and Essays*. Macmillan, London (1892).
4. J. S. Rohl, *Recursion via Pascal*. Cambridge University Press (1984).
5. I. R. MacCallum, private communication (1984).