RW1023
PA1028

ON THE CALCULATION OF FACTORIAL

OR

RECURSION & ITERATION EXPRESSED AS COMMUNICATING SEQUENTIAL PROCESSES

R W WITTY

ATLAS COMPUTING DIVISION
RUTHERFORD LABORATORY
CHILTON, DIDCOT

March 11 1978

INTRODUCTION

This paper examines the relationship between iteration and recursion, an issue raised by Westfield's work on Hasal[1]. Hoare's Communicating Sequential Processes[2] are used to model iteration and recursion. Five formulations of the 'factorial' algorithm are modelled. They are the Recursive, the expanded Iterative, the compacted Iterative, the John Gurd algorithm and the Rob Witty algoirthm. The reader would perhaps have expected only two formulations, the Recursive and the Iterative. The three other viewpoints are used to argue that recursion and iteration are merely special cases of inter-process communication structures and that neither is fundemental with respect to Hoare's concept of processes. All five formulations are examined for their efficiency when executed in both 'sequential' and 'parallel' environments with finite and infinite resources.

The author has deliberately taken on the role of Devil's Advocate in this paper. He looks forward to hearing your views on the relationship between recursion, iteration, functions and processes.

THE RECURSIVE FACTORIAL

The conventional algorithm is usually expressed as:

```
fac(x):= if x≤0 then 1 else x * fac (x-1);
```

Figure 1 shows how the recursive decent proceeds and Figure 2 shows the computational ascent. The order of evaluation of the factorial is equivalent to the statement:

```
fac(x):=(((1*1) * 2) * 3) * 4) * 5 for x = 5
```

which is the optimal order of evaluation on a sequential machine according to Nakata[3].

The formulation contains a single recursive call as the last action of the function. This is 'tail recursion' and indicates that an iterative solution is possible. Contrast John Gurd's algorithm.

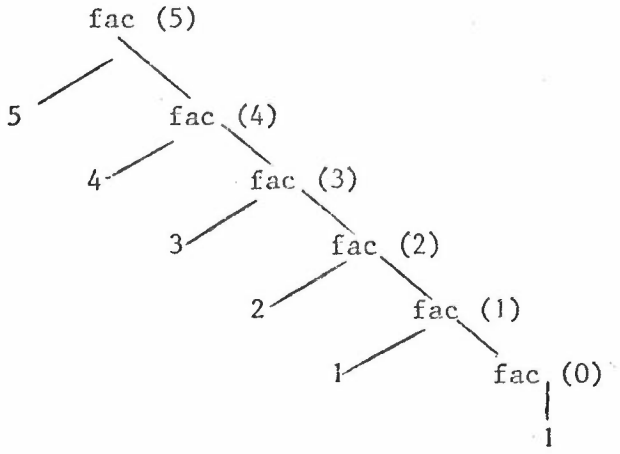


Figure 1 Recursive decent

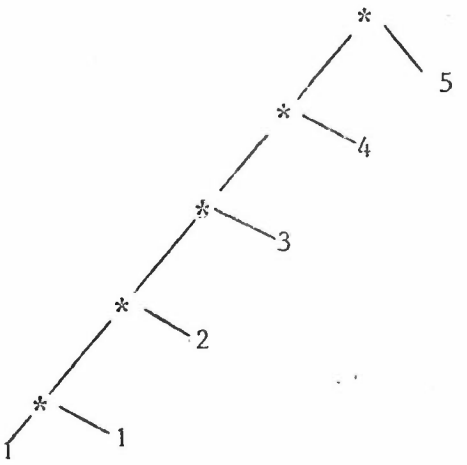


Figure 2 Computational ascent Optimal Sequential ordering

In Hoare's notation the recursion is expressed as a set of communicating processes, one process per invokation. (See general model in Figures 12, 13.)

```
Loop (0)::[USER?x ;
           Loop(1)!x ;
           Loop(1)?factorial;
           USER!factorial] ||

(i : 1.. max)Loop(i)::[Loop(i-1)?x;
                       [x ≤ 0 → Loop(i-1)!1 []
                        x > 0 → [Loop(i+1)!x-1;
                                 Loop(i+1)?fac;
                                 Loop(i-1)!x * fac]]]
```

Figure 3 Recursive Factorial

THE EXPANDED ITERATIVE FACTORIAL

It can be seen from the recursive formulation above and the general model of tail recursion in Figures 12, 13 that tail recursion is inefficient in that the result is needlessly passed back from invokation to invokation before reaching the USER. The expanded iterative formulation, Figure 4, allows each invokation to communicate directly with the USER via a merge process thus eliminating unnecessary unwinding. The formulation is iterative not recursive and models the conventional iterative formulation:

```
READ (x);
fac :=1;
i:=1;

while i ≤ x do begin fac:=fac*i ; i:=i+1 end;
```

Each iteration is expressed as a process. See the general model in Figure 14, 15. The order of evaluation is the same as Figure 2, ie optimal for a sequential machine, but now the time overhead of recursion is eliminated. Conventional programming languages cannot elegantly express expanded iteration.

```
Fac (0) :: [USER?x;
            Fac(1)!1,x;
            Merge?factorial;
            USER!factorial] ||


(i:1..max)Fac(i)::[Fac(i-1)?fac,x;
                   [i>x → merge!fac ||
                    i≤x → Fac(i+1)!i*fac,x]]  ||


Merge ::[(i:1..max)Fac(i)?factorial → Fac(0)!factorial]
```

Figure 4. Expanded Iterative Factorial

THE COMPACTED ITERATIVE FACTORIAL

The compacted iterative formulation is the conventional iterative solution and is expressed in Hoare's notation in Figure 5.

```
Fac ::[USER?x;
       fact:=1;
       i:=1;
       *[i≤x → [fact:=i*fact; i:=i+1]];
       USER!fact]
```

Figure 5. Compacted iterative factorial

The formulation is called compacted because the loop body is only stored once and reused, ie the overhead of one process per iteration is removed. The evaluation order is again optimal for a sequential machine. See general model in Figure 16, 17.

JOHN GURD'S FACTORIAL ALGORITHM

John Gurd's factorial algorithm is expressed conventionally as:

```
Fac(x):= if x ≤ 0 then 1 else fact(x,1)

fact(x,y):= if x=y then x
                   else fact(x,(x+y)/2+1)*fact((x+y)/2,y)
```

This algorithm is doubly recursive and so cannot be expressed iteratively. It generates a recursive decent tree, for x=5, as in Figure 6 and evaluation order of as in Figure 7.

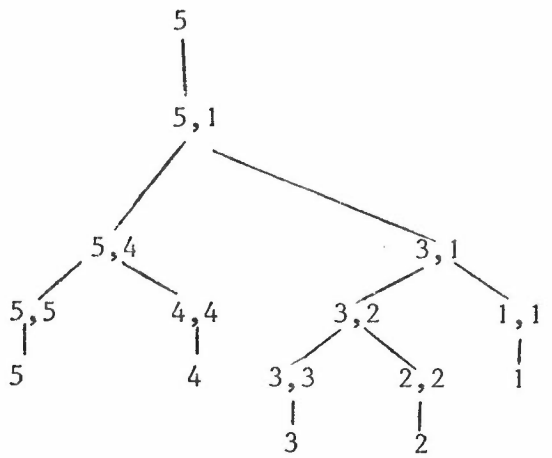


Figure 6. Recursive decent of Gurd's algorithm for x=5

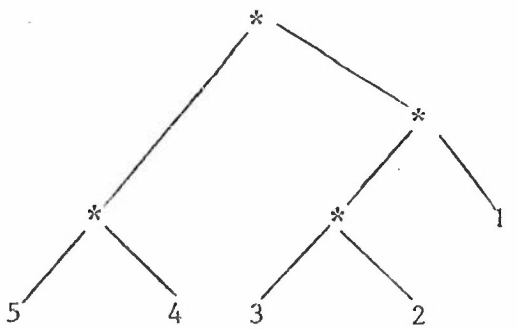


Figure 7. Evaluation order, x=5, Gurd's algorithm

The evaluation order in Figure 7 is optimal for a parallel machine with ∞ number of processors or arithmetic units according to Stone[4]. This algorithm is perhaps typical of a whole class which have a high degree of parallelism in them. If the overheads of recursive decent were small compared to the computation per invocation then this class of fomulation has much to recommend it if the parallelism can be exploited. Figure 8 shows the algorithm in Hoare's notation and Figure 9 indicates how the processes communicate.

```
Fac(0)::[ USER?x;
          [x≤0 → USER!1||
           x>0 → [Fac(1)!x,1;
                  Fac(1)?factorial;
                  USER!factorial]]]   ||

(i:1..2^max-1)Fac(i)::[Fac(i/2)?x,y;
                       [x=y → Fac(i/2)!x ||
                        x≠y → [Fac(2*i)!x,(x+y)/2+1;
                               Fac(2*i+1)!(x+y)/2,y;
                               Fac(2*i)?a;
                               Fac(2*i+1)?b;
                               Fac(i/2)!a*b]]]
```

Figure 8. John Gurd's algorithm

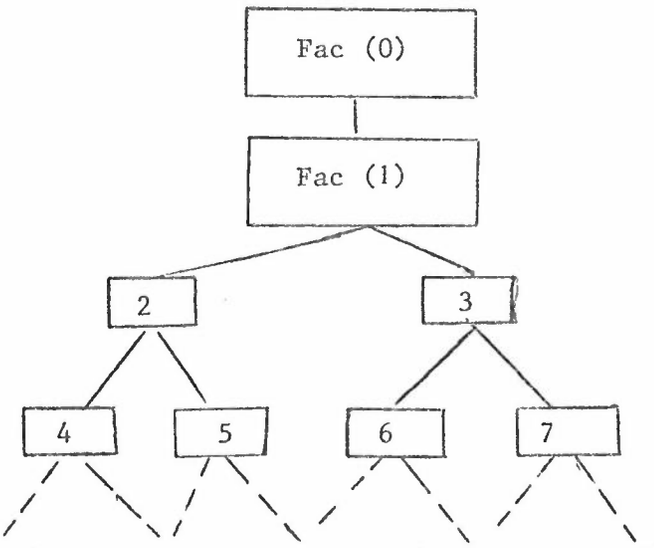


Figure 9. Process communication in Gurd's algorithm

1 ROB WITTY'S FACTORIAL ALGORITHM

Iterative and recursive formulations are traditional solutions. Hoare's notation allows a solution which is neither iterative nor recursive in principle. Consider the algorithm in Figure 1 whose processes are structured as in Figure 2. The processes form a tree-structured pipeline. The pipeline will compute the factorial in minimum time if x <= width=2**(mp-1)=number of leaves of the tree,and mp='size' of pipeline. If x > width then the loader processes feed in successive sets of values. This structure allows a finite resource to compute any factorial quickly and efficiently. It overcomes the drawback to Gurd's elegant algorithm,namely the heavy recursive overhead associated with non-trivial values of x.

This algorithm is an instance of a general algorithmic structure which is neither recursive nor iterative. It is probable that Hoare's notation will generate a flood of new algorithms exploiting novel process connection structures.

```
Fact::[USER?x;
       Starter!x;
       Fac(0)?factorial;
       USER!factorial]  ||


Starter::[Fact?x;
         [(k:1..width)Loader(k)!x]]  ||


(i:1..width)Loader(i)::[Starter?x;
                        num:=i;
                        Go:=true;
                        Stop:=false;
                        *[num<=x -> Fac(width+i-1)!num,Go;
                                    num:=num + width];
                        Fac(width+i-1)!1,Stop] ||


(i:1..(2**mp-1))Fac(i)::[Go:=true;
                         *[Go -> Fac(2*i)  ?a,contrl1;
                                 Fac(2*i+1)?b,contrl2;
                                 Go:= contrl1 and contrl2;
                                 Fac(i/2)!a*b,Go]] ||


Fac(0)::[Go:=true;
         factorial:=1;
         *[Go -> Fac(1)?f,Go;
                 factorial:=f*factorial];
                 USER!factorial]
```

Fact!factorial

Figure 1. Rob Witty's factorial algorithm

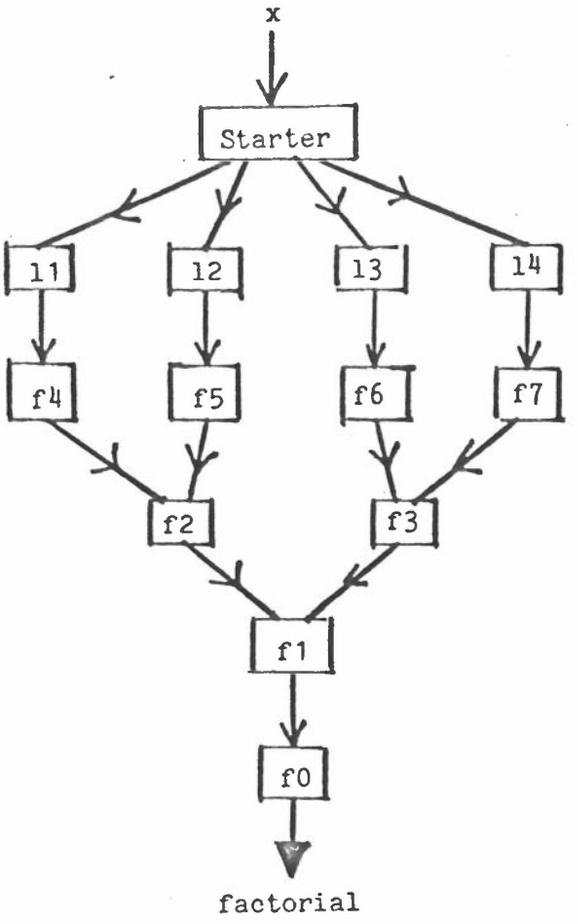


Figure 2. Process structure for mp=3.

2 ROB WITTY'S NON-DETERMINISTIC ALGORITHM

```
Fact::[USER?x; Generator!x; Controller!x;
       Controller?factorial; USER!factorial] ||


Generator::[Fact?x; j:=1;
            *[j<=x -> [Merge!j,0; j:=j+1]] ||


Merge::*[(i:1..mp)MPY(i)?f,mpynum -> Controller!f,mpynum []
         Generator?f,mpynum -> Controller!f,mpynum] ||


Controller::[Fact?maxmpys; Merge?f,mpynum;
             *[mpynum<maxmpys -> Merge?g,mpynum;
                                 Distributor!f,g;
                                 Merge?f,mpynum]
             Fact!f] ||


Distributor::[Fact?maxmpys; mpynum:=1;
              *[mpynum<=maxmpys -> [Control?a,b;
                                    [(k:1..mp)MPY(k)?free -> MPY(k)!a,b,mpyn
                                    um]
                                    mpynum:=mpynum+1]]] ||


(i:1..mp)MPY(i)::[*[true -> [Distributor!free;
                             Distributor?a,b,mpynum;
                             Merge!a*b,mpynum]]
```

Figure 3. Rob Witty's Nondeterministic Algorithm

This algorithm evaluates factorials by an essentially iterative method which exploits a fixed number of multiplier processes to increase speed. The parallelism is introduced in a way which keeps as many multipliers running as possible irrespective of their individual speeds or numbers. This is possible because the commutativity of the multiply operator enables factorials to be evaluated in a non-deterministic order.

The Generator produces the stream of integers 1through x. These are merged with partial results. Each partial result carries with it an integer which says which of the (x-1) required multiplications produced it. The Controller process exploits the fact that factorial(x) always requires (x-1) multiply operations; thus when the Merge process supplies the Controller with the result of the (x-1)th multiply the Controller terminates the calculation. The Distributor takes the next two partial results from the Controller and gives them to a randomly chosen,free multiply process, tagging them with a count of the multiplications so far. Each MPY process passes its answer to the Merge process and then signals the Distributor that it is free to perform a fresh computation.

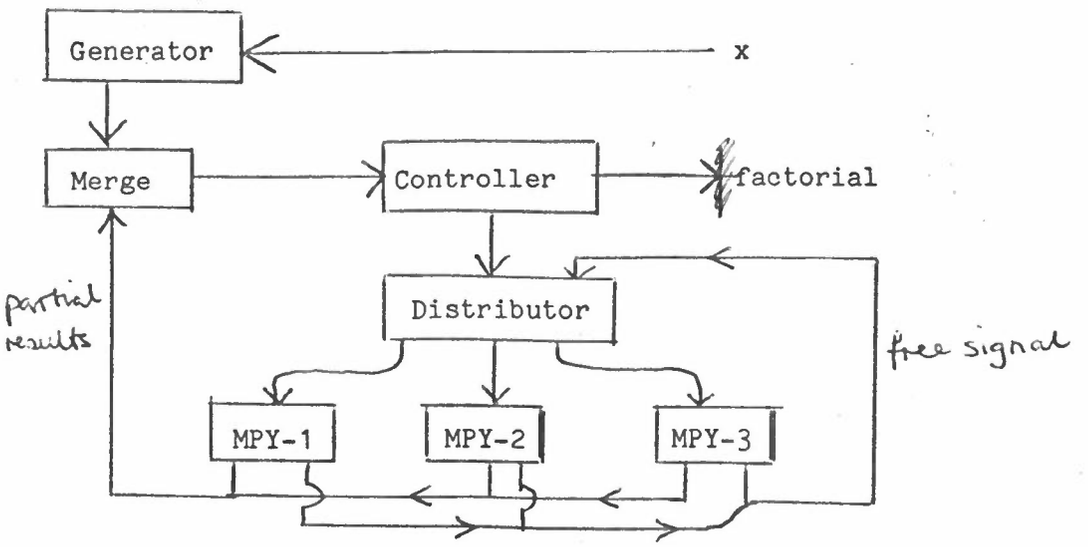


Figure 4. Process structure of non-deterministic algorithm.

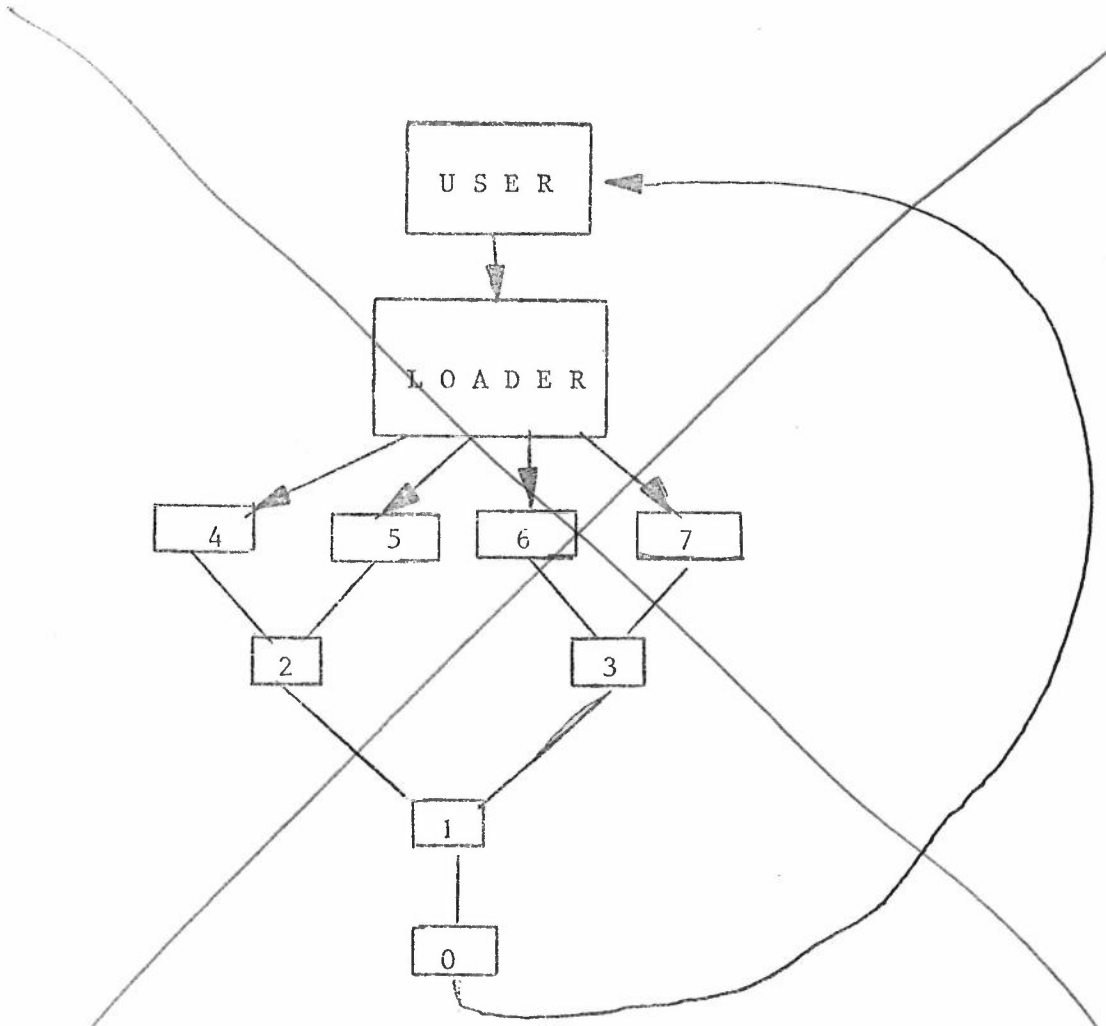


Figure 11. Witty's algorithm using 2^3 processes

GENERAL MODELS

The following seven Figures are intended to help the reader see the communication structures behind recursion and iteration. In Hoare's notation the maximum depth of recusion or the maximum number of iterations in an expanded iteration must be specified at compile time. An error occurs if a message is sent to a non-existant process. It is debatable whether this is a restriction or a feature. The author would say a feature [5].

```
Loop(0)::[ USER ? init;
           Loop(1) ! unit;
           Loop(1) ? final;
           USER ! final ]  ||

(i:1.. max)Loop(i)::[Loop(i-1)? vi;
                     [bool → Loop(i-1) ! vi ||
                      not bool → [body(i) ! vi; body(i) ? viplus1;
                                  Loop(i+1)!viplus1;loop(i+1)?fin;
                                  Loop(i-1)! fin]]]
```

Figure 12. Loop (v):= if bool then vi else Loop (body (vi));

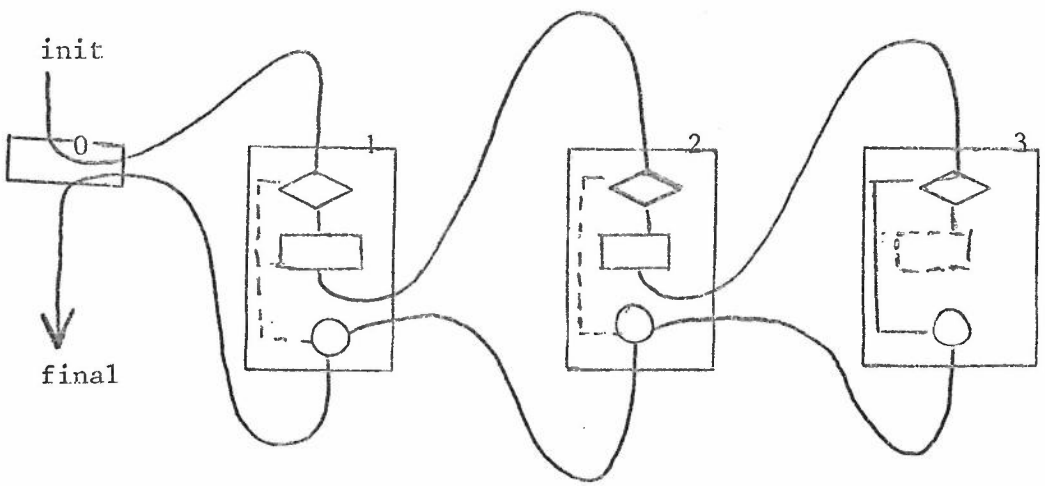


Figure 13. 'Recursive' process communication

```
Loop(0) :: [ USER?unit;
             Loop(1) ! unit;
             Merge ? final;
             USER ! final ]  ||

(i:1..max)Loop(i) :: [ Loop(i-1)? vi;
                       [ bool → Merge ! vi ||
                         not bool → [Body(i) ! vi; Body(i) ? viplus1;
                                     Loop(i+1)! viplus1]]
```

Figure 14. Expanded Iteration in Hoare's notation

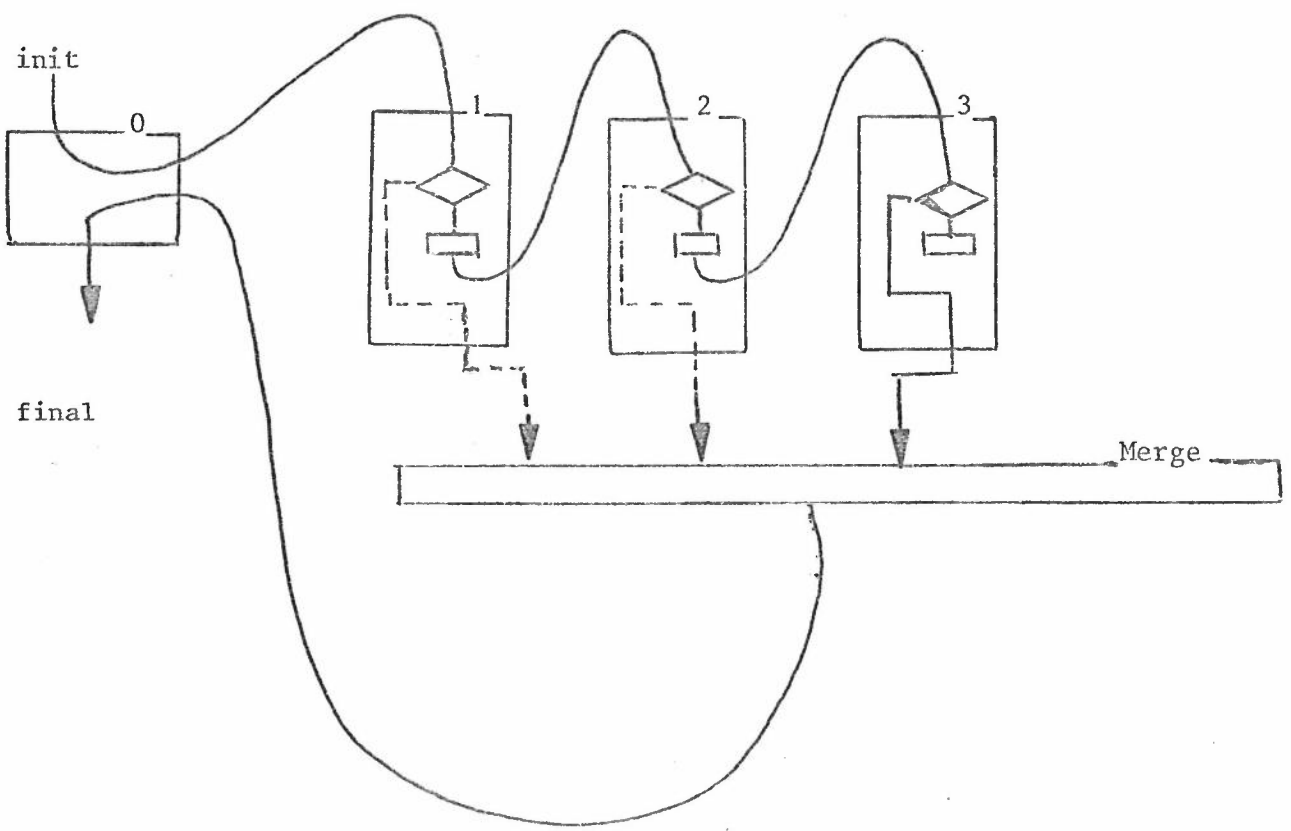


Figure 15. Process communication in Expanded Iteration

```
Loop :: [ USER?init,max;
        vi:=init; i:=1;
        *[bool and i ≤ max → [Body ! vi ; Body ? vi ; i:=i+1]
          bool and i > max → error action]
          User ! vi]
```

Figure 16. while bool and i ≤ max do begin vi := body (vi); i:=i+1 end;

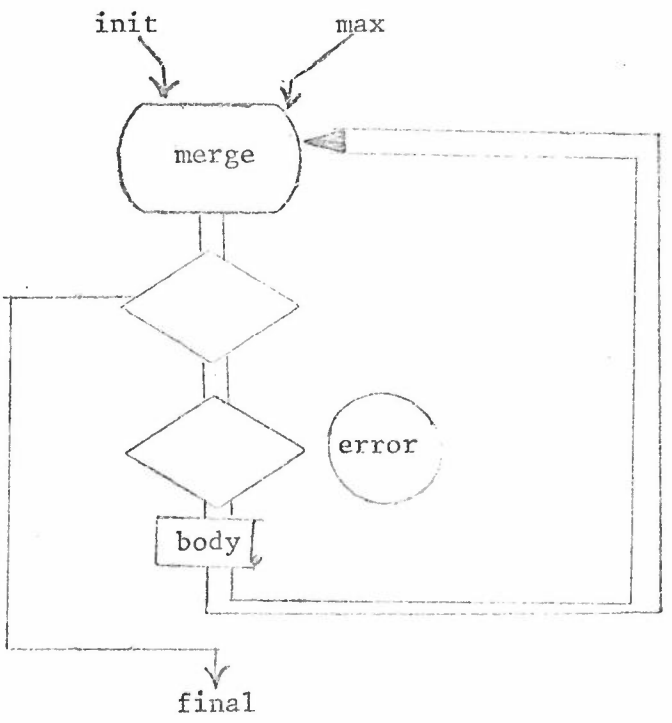
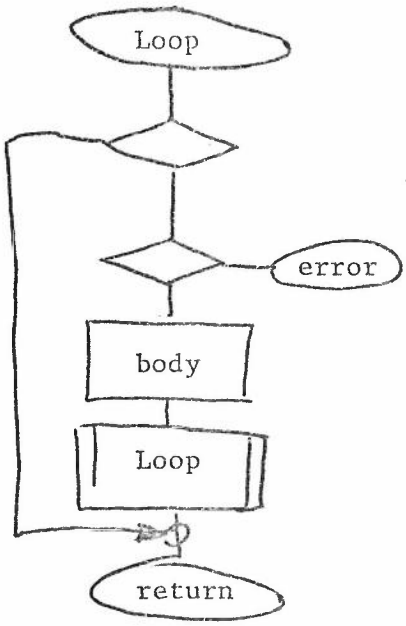
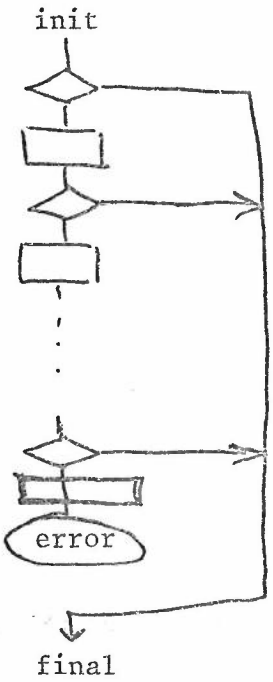


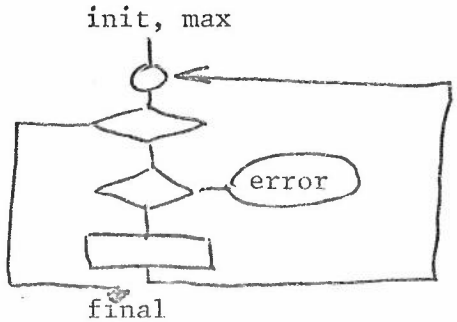
Figure 17. Conventional, compacted iteration



TAIL RECURSION



EXPANDED ITERATION



COMPACTED ITERATION

Figure 18. Flowcharts of General Models

CONCLUSION

This paper tackled the question, raised by Hasal, "Is iteration non-fundemental?" By considering the question from Hoare's standpoint the suprising answer is that neither recursion nor iteration is fundemental. Each is just a special case of inter-process communication. Witty's algorithm shows how considering processes as fundemental and iteration and recursion as not fundemental can lead to a new algorithmic approach to a well known problem.

REFERENCES

1. Hankin, Sharp. "An Informal Introduction to Hasal"
   Westfield College, August 1977.

2. Hoare. "Communicating Sequential Processes"
   to be published in CACM.

3. Nakata. "On compiling algorithms for arithmetic expressions"
   CACM Vol 10 p492 1967.

4. Stone. "One pass compilation of arithmetic expressions for a parallel processor"
   CACM Vol 10 p220 1967.

5. Anderson, Witty. "Safe Programming"
   to be published in BIT. 18 (1978) pp1-8