SETL Newsletter # 135A

J. Schwartz
July 12, 1974

Structureless Programming, or

The Notion of 'Rubble', and the Reduction of Programs to Rubble.

This newsletter continues the ruminations of Newsletter 135, but attempts to pursue more fundamental considerations. It relates also to the discussion of trans-SETL dictions, to which NL 133, 133A, and J. Earley's Berkeley report on optimisation by iterator inversion also belong.

Our thesis in this newsletter is as follows: Programming languages of 'ultra high' level, i.e., of a level substantially higher than that of SETL, must and will aim at dictional forms in which the structured interrelationships which characterise SETL and languages of sub-SETL semantic level are eliminated. In developing these languages one will therefore aim, not to support 'structured programming', but to make possible a type of 'structureless programming' whose dictions begin to be as fragmentary and disconnected as those of natural discourse.

On the scale of structure which runs from the highly structured to the totally structureless, we may, in order to fix our ideas, distinguish four typical structure-types: the *cobweb*, the *tree*, the *pipeline*, and the *rubble*. A rubble consists of mutually unrelated fragments. A pipeline consists of sections which must fit together, but each of which relates only to its immediate serial context, so that a pipeline can be assembled in serial order and its complexity does not grow with its size. In a tree, the context into which each section needs to be fitted is more complex, but complexity of local context remains bounded as the tree grows;

however, a serial order of construction is no longer possible, so some more involved, perhaps top-down, order of growth must be used. The patterns of connection in a cobweb can grow complex without bound. In building a web, sections will often have to be fitted into already-built contexts which imply difficult constraints. Moreover, a change to one part of the web may make complicated and extensive changes to other parts of the web necessary.

Programs of every one of these structural forms exist. Cobweb programs are of course familiar to all who have programmed. A rubble program is one in which program elements are unrelated to each other, can be freely added and removed, and for which each program-output can be ascribed to some individual program element rather than to the cooperative action of many interrelated elements. A hypothetical general-purpose concordance generator whose 'programs'are simply lists of words, and where the presence of a word causes all its occurences to be tabulated in some appropriately arranged listing, exemplifies our notion of rubble program.

Rubble programs are maximally easy to develop and to debug. Moreover, they are exceptionally easy to adapt to new uses. In developing such a program, one simply adds elements until all desired effects have been obtained, perhaps subtracting elements whose effects are undesired. A rubble program is correct if its individual elements are correct.

Programs of pipeline structure can be almost as easy to develop as rubble programs. One builds them section by section in serial order, conforming each newly added section to the item which preceeds it. However, an error in a pipeline program will have far more catastrophic effects than a similar error in a rubble, since all the pipeline sections past the point of error will fail to function. Moreover, when any single section section of a pipeline program is modified very many subsequent sections may have to be modified as well.

Related remarks concerning programs of tree and of cobweb structure could be made, but since the characteristics of programs of these types are generally familiar we shall refrain from making them.

One knows that one has devised 'the right' language for treating a given problem when the problem can be solved in the language by a rubble program. Beyond its description by a rubble there will in many cases be no simpler description of the problem; with a rubble, one will have reached a level at which the elemental action of the mind has become directly manifest.

Structured complex programs which solve a problem may be considered to arise from the problem's defining rubble by a process of optimising transformation (the transformed program can of course be vastly more efficient than the rubble program which underlies them.) The ideal of programming language design is to allow the programmer to express himself directly in rubble form, leaving it to an automatic optimisation system to transform this rubble into a complex structure which realises the same effects as the rubble but does so much more efficiently. However, presently available optimisers handle only limited classes of superficial transformations. Currently, therefore, the programmer is himself required to take over much of the work of optimisation, and to express himself in dictional structures complicated, by manual optimisation, to the point at which they generally approach the threshold of humanly sustainable complexity, a threshold that is rather low. Indeed, through custom optimised expression becomes the programmer's fixed habit, and the tendency to function at high complexity levels an unspoken point of professional pride. These are the reasons why programming is currently a difficult, slow affair.

An additional example will illustrate the points just made. A business application system is a programmed model of the business in which it is to be used. Written in an appropriate language, such a system will be a rubble, each of whose elements will describe the rule according to which some kind of exogenous or endogenous event is to be handled when it comes to the attention of some person or group within the firm being described. These rules are interrelated logically, but only in a loose way, and the misstatement of a few rules will produce a system that malfunctions slightly rather than catastrophically. It is a rubble of this kind which underlies each business application program of the type ordinarily seen. The program arises from the rubble when characteristic optimisations are imposed. For example, one typically groups together all processes which deal with the direct and indirect consequences of certain classes of exogenous events, thus allowing incoming 'transactions' to be processed serially to completion. The data files which must be consulted during transaction processing can then be kept in special arrangements which makes access to needed records efficient and predictable; incoming groups of transactions can be pre-sorted into an order determined by the pattern of file accesses which they will collectively require; etc.

By imagining an appropriate language, i.e., one making it easy to express rubble programs directly, we can clarify the nature of the optimisations evisaged in the preceding paragraphs.

For describing business applications, an appropriate language would be one allowing the definition of systems of inter-communicating 'clerks', each with one or more in-baskets, and with some simple rule of procedure. We imagine all the clerks to act in parallel, sending messages to each other; all share the use of certain centralised files and access certain central data objects. Certain of the clerks are sensitive to the time and date, and periodically emit messages. In such a language, a business application system can be described by a rubble of statements (in fact, not quite by a rubble, but by statements much less hightly connected than are the statements of an ordinary program). A few fragments from such a rubble serve to illustrate what is meant:

```
order-receipt: whenever exists order in input
               check name = customer-name(order) and address =
                         customer-address(order) filled-out
               ifnot finish by send <customer-info-defective, order>
                                                  to wrongorder-clerk
               check record = customer-record(name=name
                                       address = address) exists
               ifnot finish by send <unknown-customer, order>
                                                 to wrongorder-clerk
/* else */     finish by send <order, record> to order-classification-
                                                              clerk
billing-clerk: whenever exists invoice in input
               let    itemlist = item-list(invoice)
        let    total = sum price(item) over all item in
                     item-list(invoice) such that
                                    available(item) is true;
        let    total-tax = sum price(item) * taxrate(category(item))
                     over all item in item-list(invoice) such that
                     available(item) is true,...(and soforth)
```

Control transfers are largely absent from this code, which is close to a rubble, e.g., the *billing_clerk* section is activated by the receipt of an invoice and not by transfer of control from some prior code section. A program like the above can continue to function even if one of its sections, e.g. *wrongorder_clerk*, is defective or missing: the system can simply generate an input box for each missing section and accumulate items sent to it.

To transfer a rubble of this kind into a business application program of acceptable efficiency, several successive optimising transformations must be applied. The code fragments receiving copies of a given data object must be found and one must choose an execution order allowing the elimination of as many copy operations as possible. This execution order should be such as to allow the files which must be accessed during the processing of a transaction to be accessed efficiently: for example, it is desirable for code fragments accessing the same record of a file to be grouped together. Loops implied by calculations involving composite objects should where possible and appropriate be 'jammed' together to diminish the number of times that particular data items need to be accessed. Auxiliary data structures, as for example auxiliary indices, should be defined and the code needed to keep these structures current should be inserted into the developing code. Expressions E implying extensive calculation should where possible and appropriate be 'reduced in strength', i.e. kept current by inserting small adjustments of E's calculated value at each point at which a variable appearing in E is modified (this is J. Earley's method of 'iterator inversion'). Through the manual or automatic application of these and other optimisations, a tightly interconnected logical cobweb will evolve from an initial rubble R.

It is worth noting that the rubble underlying programs of business application type lies closer to the surface than is the case for programs of other types. This is because for such programs the transformations which produce an application from its defining rubble are generally of routine rather than of highly specialised mathematical character. It is this consideration that justifies the decision of several currently active 'automatic programming' groups to study systems of business application type rather than programs of some other kind.

We emphasise once more that the 'right' language for the statement of a program P has been found and that P has been given its 'right' expression in this language when P appears as a rubble. The linguistic form in which one describes the seperate fragments of the rubble is a secondary issue. It need not be harder to define a rubble in an appropriate formal language than it would be to define it in natural language. For this reason, the natural language emphasis which characterises a certain amount of current work in automatic programming can be questioned; this emphasis can be regarded skeptically as a complicating distraction from other more central problems of semantics and optimisation.

In addition to the constructions introduced into programs by optimiser actions which can be regarded as relatively routine, constructions of a different, distinctly mathematical, character will appear in programs. We regard a construction as *routine* if it is justified by assertions of predictable form which can be generated by processing families of statements whose examination is predictably profitable. On the other hand a construction is *mathematical* if it can only be justified using some fact found by good fortune within an area too enorous or disorganised to be profitably subject to systematic search.

Mathematical constructions can be reduced to rubble only as *problem statements*, not as *algorithms*. Here we distinguish problem statements from algorithms by the fact that they make reference to objects too vast for actual construction, and to searches and processes of selection extended over these vast objects.

As an example of this distinction, consider the notion of sorting in its relationship to the algorithms actually used for sorting. To define the notion of sorting in an 'algorithm free' way we can proceed as follows: an n-permutation is a 1-1 map P from the set $\{i, 1 \le i \le n\}$ to itself; given two vectors v and $\bar{v}$ of length n, $\bar{v}$ is said to be in the *permutation range* of v (we write $\bar{v} \in$ permrange(v)) if there exists a permutation p such that $\bar{v}(i) \equiv v(p(i))$. To sort v is to find a $\bar{v}$ in its permutation range such that $\bar{v}(i) \le \bar{v}(i+1)$ for all $1 \le i < n$. What we have just given is a problem statement rather than an algorithm since the collection of n-permutations contains n! elements and is thus far too large to be searched explicitly. To obtain an algorithm from this problem statement one transforms it mathematically using a method which may be described abstractly and generally as follows: An object x satisfying a predicate C(x) is to be found within a set s which cannot be searched explicitly, either because s is too large or because it is expressed in terms which make s very difficult to compute. To construct x, one chooses some initial object x_o in s, and finds a transformation f of s into itself which has the property that $f(x) = x$ implies C(x). Then one generates the sequence $x_o, f(x_o), f^1(x_o), \ldots$, If f has been chosen appropriately, this sequence will stabilise; and the first element $f^n(x_o)$ satisfying $f^n(x_o) = f^{n+1}(x_o)$ is the desired x.

Many variants of this paradigm will occur. It may for example be convenient to embed s in some even larger set t, and to use an auxiliary transformation f which maps t into t, but where $f(x) = x$ implies $x \in s$. One may make use of an auxiliary predicate $C'(y)$ for which $C'(x_o)$ holds and for which $C'(y)$ implies $C'(f(y))$; then one need only prove that the two propositions $C'(x)$ and $f(x) = x$ together imply $C(x)$. A predicate C' with these properties is said to be a *continuing assertion* of the iteration $x_o, f(x_o), f^2(x_o),\ldots$. The target predicate $C(x)$ may be decomposable as a conjuction $C_1(x)$ <u>and</u> $C_2(x)$; in this case, one can try to find two transformations f_1, f_2 of s into itself, such that $f_1(x) = x$ implies $C_1(x)$, such that $f_2(x) = x$ implies $C_2(x)$, and such that $C_1(y)$ implies $C_1(f_2(y))$. When these are found, one can select x_o in s, carry the sequence $x_o, f_1(x_o), f_1^2(x_o),\ldots$ to convergence to obtain an element x'_o, and then carry the sequence $x'_o, f_2(x'_o), f_2^2(x'_o),\ldots$ to convergence to obtain the desired x.

We see from the above that set-theoretic expressions which use unpleasantly large sets as intermediate terms in the definition of objects of more readily calculable size are replaced by *while* or *until* loops which construct these objects in far more efficient ways. We shall call this process *mathematical expansion*, and speak of the loop as arising from the mathematical expansion of the set-theoretic expression which underlies it. Within a loop arising in this way loop subsidiary set-theoretical expressions may occur, and these will themselves expand into *while* loops, nested to some modest depth.

To be solved, a mathematical problem P must first be recognised, and must therefore have a set-theoretical statement which is not too complicated.

An algorithmic solution of P is obtained, first by restating it in more advantageous but still not vastly complicated set-theoretic terms, and then by progressively transforming its statement into an algorithm. Hence we expect most mathematical algorithms to consist of nested sets of *while* loops ultimately containing elementary set-theoretical expressions. Inplicit in such a program is a tree (we shall call it the *determining tree* of P) whose nodes are the set-theoretic expression which the while-loops of the program realise. Trees of this type can be developed directly by the mathematical activity of the mind, the mathematical expansion of each node leading, in a manner isolated enough to be comprehensible, to the generation of a few descendant nodes. The determining tree T of a program should be loosely reflected even in the loop structure of the program's final form and the overall structure of T should therefore correspond to the structural facts which interval analysis of the program will reveal. Note that we consider each loop L in a mathematically flavored program to realise some underlying set-theoretical expression E which the loop is contrived to evaluate; E defines the 'meaning' of L and the role that L plays in any larger loops in which it may be embedded. By progressively reconverting each loop L of a program P into the E from which L arises by mathematical expansion, we make explicit the strategic approach used to develop P, and ultimately reduce each P of mathematical character to the definitional statement from which it was generated. This latter statement may in turn be a fragment of some rubble in which the mathematical algorithm P is used as a device.

The way in which we choose to expand a set-theoretical expression E into a loop will depend on the context of facts within which E is to be evaluated.

For example, to find the smallest component of a vector v which exceeds a given quantity x requires a full search of v in the general case, but only a binary search if v is known to be sorted. Consequently, it will sometimes be advantageous in transforming a set-theoretic expression E into a loop to construct a loop L within which set-theoretic expressions E_1 just as complicated as E or even identical to E appear, provided that the context of assertions available inside L is substantially more advantageous that the context in which L itself appears. This makes it plain that a set-theoretical expression E is not a full description of the algorithm which realises it. Generally speaking, the cost C of evaluating a set-theoretical expression E will be a function both of E's parameters and of the context of facts within which E must be evaluated; and C can depend very sensitively on this context. If E_1 occurs inside a loop L, then the total cost of its repeated evaluation will be the cost of a single typical evaluation times the expected number of times that L is executed, which minimised (in a manner taking advantage of the fact-context in L) gives the cost of evaluating E in its context. Repeating this calculation recursively for all the nodes of the determining tree T representing an algorithm under development gives the expected efficiency of the algorithm. The essence of algorithm design is to structure T in such a way as to guarantee each significant expression E appearing in T a surrounding fact-context allowing E to be evaluated in an especially efficient way.

The determining tree of a program P serves also as a guide to the construction of a proof of P's correctness. To build such a proof, one will aim first of all to show that each loop L in P does realise the set-theoretical transformation which it is meant to realise.

This fact will constitute the core clause of L's *output assertion,* which must be shown to be true on exit from L. To prove this output assertion one will require an *input assertion* giving facts known to be true on entrance to L; in addition, a *continuing assertion* steadily valid within L will be used. The output assertion of each loop L must be compatible both with the input assertion of any loop L' which follows L and with the continuing assertion of the loop $\bar{L}$ including L if L is not 'outermost'. The determining tree of P, taken with the various assertions hung on the nodes of P, is what we call the *annotated determining tree of P,* and describes the mathematical content of an algorithm P of mathematical type completely. That part of a programmer's work which lies at the design level consists in the development of this tree; the rest can be regarded as the manual application of routine optimisations (which application may of course still be quite difficult to accomplish.) An ideal language for the statement of mathematically flavored algorithms would be one which allowed the annotated determining tree of algorithms to be stated directly, and which itself evolved programs from these trees.

Note that the items which appear in a program's annotated determining tree do not share the dynamic character of the program but have a purely static set-theoretic character. Experience shows that static, tree-like constructs tend to be relatively error-free; for example, expressions, including complex set-theoretic expressions, can be written with a lower probability of error than even rather simple loops. It is also instructive to make the technical remark that formal proofs of program correctness will be subject to a minimum of irrelevant complication if the language in which one writes the programs which are to be proved correct is semantically and syntactically identical with, or at least a sub-language of, the language in which the correctness proofs are to be given.

Since set theory is very likely to be the language in which all but very simple proofs are couched, this remark serves to justify SETL. A similar remark justifies SETL's decision to avoid pointer semantics entirely: a language in which the final instruction of the code sequence

```
x = 0;
put x in s;
...                         (2)
x = x + 1;
```

changes s can be massively irritating to the would-be correctness prover. Indeed, it is hard to see how programs written in a language having this character can be proved correct except by re-expressing their semantic intent in explicit set-theoretic terms, i.e., reprogramming them in a manner much like that which would be used if they were to be transcribed into SETL.

The well-known bubble sort algorithm furnishes a very simple illustration of the general points made in the proceeding pages. We will find it convenient to write this algorithm, as well as a few of the other algorithms to be examined later, using an *until* loop construction of the form

(1) (until C) *block*;

where C denotes a boolean expression and *block* a block of code. Semantically, an *until* loop is executed until either the condition C becomes true(which we call termination by success) or the execution of *block* is seen to be without effect, (which we call termination by futility). If C is expressed using one or more universal quantifiers involving one or more parameters $x_1,\ldots x_n$, then each time *block* is executed a set of parameter values making C false will be supplied to *block*.

An advantage of the construct (1) is that when the loop (1) is terminated by success the condition C is known to be true as an output assertion.

With these conventions we may write the bubble sort as

```
(2)  u = v;
     ( until 1 ≤ ∀n < # u|u(n) le u(n+1))
                                  <u(n), u(n+1)>=<u(n+1),u(n)>;;
```

The input assertion is that v is a vector of reals; in order that (2) should be a sorting routine, we require that

```
(3)  u ε permrange(v) and 1 ≤∀n < # u|u(n) le u(n+1)
```

should be an output assertion of (2). But the second clause of (3) is simply the condition appearing in the *until* clause of (2); and nε permrange(v) is easily seen to be a continuing assertion of this *until* clause.

More conventional bubble sort algorithms arise from (2) by the application of relatively routine optimisations. Note in particular that evaluation of the '∀'-quantified expression C in (2) involves a search loop which can search indices in ascending order; and that immediately after finding a first n violating C and performing the interchange which this implies we can be sure that u(j) le u(j+1) for j<n-1. This allows us to rewrite (2) in a conventional form as

```
(4)  u = v;
     n = 1;
     (while n lt # u)
            if n eq 0 then n = 1;;
            if u(n) le u(n+1) then n = n + 1;
                  else <u(n), u(n+1)> = <u(n+1), u(n)>; n = n - 1;;
     end while;
```

2. The annotated determining tree of a more substantial mathematical algorithm: Floyd's *heapsort*.

For a more substantial example of the process of development which we take to underlie programs of mathematical type, we consider R. Floyd's *heapsort*. We shall develop this algorithm in top-down form. The algorithm has a vector v of reals as input. It is required to be a sorting routine, i.e. to have

(1) $u \in$ permrange(v) and $1 \le \forall n < \# u \mid u(n)$ le $u(n+1)$

as an output assertion. The problem statement (1) is what we aim to optimise by a process whose first stages are manual but which becomes automatic as soon as possible. As the algorithm's first form we take

(2)
```
        /* v is input */
        u = nult; y = v;
        (until y eq nult)
              y = minbot y;
              u(#u+1) = y(1);
              y(1) = y(#y);
              y(#y) = Ω;
        end until;
```

Here minbot is a subsidiary transformation, for which an algorithm must still be given; we require this transformation to have the output proposition

(3) (minbot y) $\in$ permrange(y) and $1<\forall n\le\# y\mid$ (minbot y)(1) le (minbot y)(n).

Given this fact concerning minbot, it is not hard to see that

(2) has (1) as output assertion. Indeed, the *until* loop of (2) has

(4) u + y ε permrange(v) and 1 ≤ ∀n < # u | u(n) le u(n+1)
and 1 ≤ ∀m ≤ # y |if u eq nult then t else u(#u) le y(m)

as a continuing assertion: This assertion is clearly true on entrance to the *until* loop since on loop entry u eq nult and y eq v; in view of the output assertion (3) of the transformation minbot, the body of the *until* loop of (2) preserves the assertion (4). On loop exit we have y eq nult, and therefore (1) results from (4).

Now we must realise the transformation minbot. For this, we can use the following code:

```
(5)                      /* y is input and w output */
    w = y;
    (until 1 ≤ ∀n ≤ # w/2|w(n) le w(2*n) and
                         if 2*n+1 gt # w then t else
                                        w(n) le  w(2*n+1))
    x = if (2*n+1) gt # w then 2*n else if  w(2*n) le w(2*n+1)
                                       then 2*n else 2*n+1;
    <w(n), w(x)> = <w(x), w(n)>;
end until;
```

If the *until* condition of (5) is satisfied, then the second clause of (3) is satisfied as well, since if not the minimum component of w would have an index m different from 1, hence of the form 2*n or 2*n+1, and this would violate the *until* condition. On the other hand, if the *until* condition is not satisfied, then either w(n) gt w(2*n) or w(n) gt w(2*n+1); and then w(n) gt w(x) is certain, so that the permutation <w(n), w(x)> = <w(n), w(x)> changes w.

This makes it clear that the *until* loop of (5) cannot terminate until the second clause of (3) is satisfied. On the other hand, the loop clearly has w ε permrange(y) as a continuing assertion; thus the first clause of (3) is also an output assertion of (5).

By substituting (5) into (2), we therefore obtain a complete SETL algorithm having (1) as output proposition. However, the efficiency of this algorithm can be improved considerably by applying a few transformations to it. Suppose that the input vector y of (5) satisfies

```
(6)        1 < ∀n ≤ # y /2 | y(n) le y(2*n) and
                if 2*n+1 gt # y then t else y(n) le y(2*n+1)
```

Then it is not hard to see that (if we insert   x = 1 at the beginning of (5)) the *until* loop of (5) will have

```
(7)        1 ≤ ∀n ≤ # w/2 | if n ne x then (w(n) le w(2*n)
                       and if 2*n+1 gt n then t else
                             w(n) le w(2*n+1))
```

as a continuing assertion. This makes it plain that if (6) is satisfied the code (5) will produce the same output w as the code

```
(8)        w = y; x = 1; fixedup = f;
           (while 2*x le # w and not fixedup)
                n = x;
                x = if (2*n+1) gt # w then 2*n
                    else if w(2*n) le w(2*n+1)then 2*n else 2*n+1;

                if w(n) le w(x) then fixedup = t; else
                          <w(n), w(x)> = <w(x), w(n)>;;
           end while;
```

A similar argument shows that if the input vector y of (5) satisfies

```
(9)             1 ≤ ∀n ≤ (#y-1)/2 |y(n) le y(2*n) and
                     if 2*n gt # y then t else y(n) le y(2*n+1)
```

then (5) will produce the same output as the code

```
(10)            w = y; x = # y;
               (while if (x/2) eq 0 then f else w(x/2) gt w(x))
                     <w(x/2), w(x)> = <w(x), w(x/2)>;
                end while;
```

It is easily seen using arguments like those which are given above that the code

```
(11)            w = y(1:1); z = y(2:);
                (while z ne nult)
                      w = minbot w;
                      w(# w+1) = z(1);
                      z = z(2:);
                end while;
```

realises the transformation w = minbot y. But (9) (with w subsituted for y) is a continuing assertion of (11). Thus within (11) the code (10) can be used to realise the minbot transformation. This shows that the following code produces the same output w as does (5):

```
(12)            w = y(1:1); z = y(2:);
                (while z ne nult)
                      x = # w;
                      (while if (x/2) eq 0 then f else w(x/2) gt w(x))
                            <w(x/2), w(x)> = <w(x), w(x/2)>;
```

```
          end while;
          w(# w+1) = z(1);
          z = z(2:);
     end while;
```

We have seen that any vector y = minbot z constructed by (5), or by (8) when (8) is equivalent to (5), must satisfy (6). Hence (6) will be a continuing assertion of the *until* loop of (2) if (6) is true on entrance to this loop; which can be secured by modifying (2) slightly, to make it

```
(13)      u = nult; y = minbot v;
          (until y eq nult)
                y = minbot y;
                u(#u+1) = y(1);
                y(1) = y(#y);
                y(#y) = Ω;
          end until;
```

Then in (13) we can realise minbot in its efficient form (8) inside the *until* loop of (9), and in its general form (12) outside this loop. Making the substitutions implied by this remark, and eliminating a few unnecessary variables, we obtain the following code:

```
(14)      u = nult;
          y = v(1:1); z = v(2:);
          (while z ne nult);
                x = # y;
                (while if (x/2) eq 0 then f else y(x/2) gt y(x))
                        <y(x/2), y(x)> = <y(x), y(x/2)>;
                end while;
                y(# y+1) = z(1);
                z = z(2:);
          end while;
```

```
(while y ne nult)
        w = y; x = 1; fixedup = f;
        (while 2 * x le # w and not fixedup)
                n = x;
                x = if (2*n+1) gt # w then 2*n
                        else if w(2*n) le w(2*n+1) then
                                        2*n else 2*n+1;
                if w(n) le w(x) then fixedup = t; else
                        <w(n), w(x)> = <w(x), w(n)>;;
       end while;
       y = w;
       u(#u+1) = y(1);
       y(1) = y(# y);
       y(# y) = Ω;
end while;
```

Additional improvements, having essentially the nature of conventional optimisations, can now be applied to (14) to produce *heapsort* in its ordinary form. The main observation required is that all the vectors appearing in (14) can be represented as subsections of one single vector. Applying some of transformations which this observation makes possible and a few conventional optimisations in addition we obtain *heapsort* in its final form:

(15)
```
nv = # v;
ny = 1; /* v(1:ny) will represent y;
                    v(ny + 1:) will be z */
(while ny lt nv)
        x = ny;
        (while if (x/2)  eq 0 then f else v(x/2) gt v(x))
                <v(x/2), v(x) > = <v(x), v(x/2)>;
        end while;
```

```
        ny = ny + 1;
    end while ny;
    /* now y will be v(1:ny)
              and u will be v(nu+1:)  in reverse order */
    (while ny ge 0)
        x = 1; nyo2 = ny/2;
        (while x le nyo2)
              n = x;
              x = if (2*n+1) gt ny then 2*n
                  else if v(2*n) le v(2*n+1) then 2*n
                                                   else 2*n+1;
              if v(n) le v(x) then quit; else
                      <v(n), v(x)> = <v(x), v(n)>;;
        end while;
        <v(1), v(ny)> = <v(ny), v(1)>;
          ny = ny - 1;
    end while;
```

27

In the ordinary informal sense which attaches to the word 'proof', e.g. in connection with proofs published in mathematical journals, we may claim to have proved the program (13) to be correct. Of course (13) may well be incorrect anyhow, since we have given only an informal proof of its correctness, and it is entirely possible either that some misprint has intruded itself into the text either of (13) or of some one of the program texts which led up to (13), or that some minor logical error has come into either the explicit or the implicit part of our reasoning. Generally speaking, we are only guaranteed against malfunctioning of a program which has been 'proved' correct if its correctness proof has either been generated by an automaton or stated in a formal language and verified by an automaton Without automatic verification, no stronger guarantee attaches to a proof of program correctness than attaches to mathematical proof generally, to wit, that the reader, by making 'appropriate small emendations', can very probably correct any errors which the proof may contain. In practical terms, this is not a better guarantee than that which attaches to programs developed and debugged in the ordinary way. Of course, a correctness proof for a program P serves to 'double-check' P in much the same way as would the development of a very careful set of comments for P. Moreover, adherence to the mathematical rules of proof will generally result in checks which are particularly exhaustive.

What then is the role which proofs of program correctness can be expected to play in the development of programming technique? In confronting the question, it should first of all be noted that correctness proofs developed for existing algorithms will generally be mathematically uninteresting. Indeed, as has been observed in section 1 above, an algorithm's annotated determining tree, from which the algorithm is produced by what is an essentially routine process of manual compilation, includes propositions which together constitute a proof of the algorithm's correctness. To prove the algorithm correct is therefore only to make explicit an argument which the algorithm's inventor may have left implicit; this may be a valuable expository service, but it will generally not involve anything that can claim to be a new mathematical discovery. The problem of proving programs correct is therefore a problem of pragmatic character, namely that of developing automatic or semiautomatic systems which will allow purported proofs to be stated formally and checked automatically, and which will lighten the heavy burden of preparing correctness proofs, especially for very large programs, automatically generating routine proof details. At the present time, we are far from possessing proof-generating algorithms capabable of generating proofs of a length or complexity comparable to that sketched above in connection with the *heapsort* algorithm; thus a proof verification system is in fact all that can be hoped for as a relatively near-term possibility. To be practical, such a system will have to handle assertions written in general set theoretic terms and understand the propositional implications of a wide class of program transformations. It is particularly essential that a correctness-verification system afford its user a large measure of stability, making it unnecessary for him to readjust the whole of a proof each time some modest adjustment is made in the algorithm which he is working. To develop such a system at the present time is a formidable task. Stability will of course be enhanced if algorithms are stated in a language of abstract character in which many incidental, implementation-related details are suppressed. We therefore assert that the development of correctness-proof techniques to a level of practical utility will be closely bound up with the development of high level languages and of methods for the automatic optimisation of these languages.

3. Summary.

For emphasis, we repeat our main point: a program P arises from the application of optimising transformations to a defining rubble R. Fragments of two types will be found in rubbles R: *elemental* fragments, which directly define some desired element of output or of system response; and *mathematical* fragments, which define some set-theoretic object or operation to be realised or constructed efficiently in P. Mathematical fragments are introduced into P either by manual optimisation operating in a range which lies beyond the reach of automatic optimisation procedures, or because the problem described by R has or can appropriately be given some inherently mathematical formulation. Programs are given much of their structure by the action of an optimiser acting on the essentially structureless R; what additional structure they have will generally derive from structure inherent in their input or desired output, or more generally in the data environment in which they operate.

We have projected *structureless programming*, i.e. the development of systems in which programs can be defined in rubble form and all else done by an automatic optimiser, as an ideal. What then is *structured programming*? We offer the following definition: structured programming is a technique, useful as long as optimisers of the power needed to support structureless programming are unavailable, which by imposing an appropriate discipline helps the programmer to optimise programs manually while avoiding the development of unmanageable complications.

4. Appendix: A debugging aid suggested by the foregoing.

It is suggested in section 1 that programs of mathematical type will generally consist of nests of while loops, in which each loop realises some simple set-theoretical transformation. A debugging aid which displayed the state of relevant data on entrance to and exit from each of the while-loops of a program might be useful. On entrance to such a loop, all data values to be used within the loop ought to be saved. On exit from the loop, all data values modified within the loop and alive on loop exit ought to be collected, and printed together with the data gathered on loop entrance. Excessive output will be avoided if this trace data is only printed for the first few entrances/exits made to/from each loop.

The line of argument set forth in section 1 suggests that program debugging by the insertion into code of assertions to be checked dynamically must always fail to be mathematically decisive. Indeed, the full set of assertions constituting the proof that a program is correct will generally make reference to at least a few exceedingly large objects, impossible to calculate explicitly.

Bibliographic Note:

A proof of the correctness of *heapsort* was first given by Ralph London in *Proof of algorithms: a new kind of certification (Certification of Algorithm 245 TREESORT 3)*. CACM 13, 6, June 1970, pp. 371-373. The proof offered above is of course very much like London's, but our intent is somewhat different from his since he aims to annotate an existing code whereas we have been at pains to emphasise the guiding role which an implicit proof plays in the genesis of an algorithm. For a good recent survey of literature on program correctness proofs, see London's *The Current State of Proving Programs Correct*. Proc. ACM 25th Annniversay Conference, August 1972.