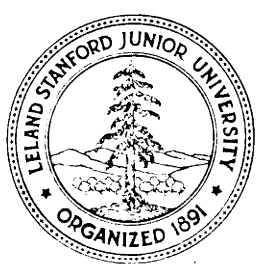
A REVIEW OF "STRUCTURED PROGRAMMING"

by

Donald E. Knuth

STAN-CS-73-371
June 1973

COMPUTER SC I ENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UN IVERS ITY



A Review of "Structured Programming"

by

Donald E. Knuth

Abstract

The recent book Structured Programming by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare promises to have a significant impact on computer science. This report contains a detailed review of the topics treated in that book, in the form of three informal "open letters" to the three authors. It is hoped that circulation of these letters to a wider audience at this time will help to promote useful discussion of the important issues.

This research was supported in part by the National Science Foundation under grant number GJ-36473X, IBM, and Norges Almenvitenskapelige Forskningsråd. Reproduction in whole or in part is permitted for any purpose of the United States Government.

STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

COMPUTER SCIENCE DEPARTMENT

Telephone:
415-321-2300

December 7, 1972

Prof. Dr. E. W. Dijkstra
Mathematics Department
Technological University
Eindhoven, The Netherlands

Dear Edsger:

Ole-Johan Dahl has just given me a copy of the new book Structured Programming, and I want to congratulate you on an especially fine job. It is delightful to see these important ideas in print at last, and the book will no doubt prove to be extremely influential. Your unique style of writing holds the reader spellbound; it's the kind of book you can't put down until you reach the end, and once you have read it your life is not the same thereafter.

Of course, I don't agree 100 percent with everything you said -- this is inevitable whenever artistic or aesthetic judgments are involved -- so I'd like to jot down some of my current feelings on these issues in hopes of further clarifying and perhaps strengthening the ideas. In other words, I am now writing a letter to myself based on your letters to yourself. And I hope you will have time to read mine as I have read yours.

Here then are specific comments prompted by your Notes on Structured Programming, cross-referenced by the page number where they appear in the book.

The first time I felt like raising a mild protest was on page 7, where I didn't realize that dd and r were intended to be integer variables. (This is of course explained nicely on page 15, but I didn't know it at the time.) The program makes an instructive example with respect to floating-point computations also. In the first place, when floating-point operations are properly rounded, the program does in fact leave (1) invariant; but if truncation arithmetic (IBM's style) is used, it fails. For example, in decimal notation, if dd = 2 .0000001 and r = 2.0000000 , the program first sets dd := 1.0000000 . On nearly all computers which do rounded arithmetic, your proof of the invariance of (1) is apparently correct; nevertheless, after the program is repeated 100 times the result will be to set d to zero while r remains positive! The reason is the preposterous convention, almost universal, of replacing 'exponent underflow' by a zero result with no interruption of the program. Perhaps examples like this will finally be able to convince hardware designers that it is absurd to destroy all the significant figures without warning.

However, you probably aren't interested in such things, so I shall move on. The "primes" program has come out very nicely, since I last saw it; I'm glad the rather uninformative digression about "throdd" numbers has been suppressed, and the instructive mult table is a welcome addition because it sheds light on redundant data structures where the invariant need not completely characterize the redundancy. I only wish to comment on two things regarding this program: First, on page 37, where you begin to assume that remainders cannot be computed conveniently, the reader who has been following carefully will recall that an efficient and elegant algorithm for computing remainders has been presented on page 13 (the computation on q may be suppressed). So this is the obvious thing to do in 2b4(4)d , and it is very instructive to observe that it is much worse than the mult table since the latter requires comparatively little space. The moral of the story is clearly to avoid using operations blindly without considering the context, as you point out briefly later in Section 11.

My second comment has to do with this question of context. Of course the nomenclature "2b4(4)d" is not pretty, but the real problem as we get to the end of the primes program is conceptual, not notational. The steps of program construction have unfolded very nicely and naturally but at the end we can't really fold them together again -- we seem to be almost looking at the entire program as a whole, with it all in our head at once. Thus when the reader gets to page 38, with the final 'patches' on level 2b5(4) , it becomes suddenly much harder to understand what is going on.

Perhaps it is inevitable that a programmer must in fact have reached a conception of the whole program at once (in an appropriately structured form of course) by the time he has finished. But this seems to imply that the difficulty of programming increases greatly with the size of the program, while ideally we would like the level of difficulty to remain workable. The same phenomenon occurs in the picture-drawing program; by the time the reader/programmer gets to the end of LINER at the bottom of page 56, he needs to be flipping pages back and forth and essentially keeping the status of the whole program in his head. The latter may almost be necessary; I have always had a feeling that a talent for programming consists largely of the ability to shift quickly from macroscopic to microscopic views of processes.

Some evidence that you yourself are in fact keeping the entire context of the program in mind appears on the top of page 54, where you say fifty blank lines would be output, while we are at this point only looking at the 'build' procedure which purports to be independent of the output. I don't criticize you for this (although this particular instance is unnecessary context); I merely want to illustrate what seems to be a part of programming psychology. You say rightly on page 50 that we must tie loose ends together again.

Herein lies the dilemma, and the conflict. A notation which expresses the separation of tasks nicely eventually gets into difficulties when there are many loose threads running through a single pearl. Yet the pearls are valuable as a means of coordinating the individual design decisions.

Perhaps the new display terminals will provide the answer, as a notation for programs that will provide an appropriate reflection of the structure as we might have it in our minds. You have perhaps had a dream much like mine: Wouldn't it be nice to have a glorious system of complete maps of the world, whereby one could (by turning dials) increase or decrease the scale at will? A similar thing can be achieved for programs, rather easily, when we give the programs a hierarchic structure like those constructed step-wise. It is not hard to imagine a computing system which displays a program in such a way that, by pressing an-appropriate button, one can replace the name of a routine by its expansion on the next level, or conversely.

The independent design decisions (i.e., the pearls) could be identified in such a system by marginal classification codes, saying which pearl each line belongs to. An example appears below. This seems to me to be a reasonable way to bring the appropriate context into each pearl, yet retaining the pearl's identity. (We need another word for pearl, though; what should it be'?)

The viewpoint about a sequence of machines and operation codes, which you have nicely expressed in Section 13, has often proved useful to me also in a more explicit form where the data itself gets transformed into instructions for a pseudo machine. An example of this appears in the appendix to my paper on computer-drawn flow charts, Comm. ACM 6 (1963), 555-563 . A two-pass algorithm is described, wherein the first pass encodes the data into a pseudo machine language and the second pass executes that program. If you have the time, I hope you can read this old paper of mine, even though I was of course much more naive at the time; it seems to me that a number of important principles are involved, and that a closer study of such an algorithm may lead to increased understanding of program construction (and proofs).

I am not part of your audience who were "deeply troubled" by the time the top of page 56 was reached. But I must admit to being deeply troubled at the top of page 57, and not only by the fragmentation referred to above. Suddenly you had jumped to a choice of data structure very different from what I had expected. Namely, why was the possibility of representing the data as a list of 1000 pairs, sorted into lexicographic order (decreasing y , then increasing x) never considered?

After thinking about this a little, I'm convinced that it is a question of density. Consider the identical problem, but with the fx(i),fy(i) only to be printed for $0 < i < 10$ instead of 1000 .

I believe you would have reached a different solution, and it is interesting (to me) to locate the point where the solutions begin to differ, and the reasons for the difference. It seems that an appropriate way to apply your principles would be to further expand print, before going into a consideration of image, possibly thus:

```
PRINTER
begin type loc;
print: {initialize loc; repeat{advance to next position; printsymbol]
       until final loc reached];
instr initialize loc (loc), advance to next position (image, loc),
     printsymbol (image, loc), final loc reached (loc)
end
```

This seems to be a useful, if not necessary, step in the development, because it expresses what properties of image the print routine needs. Now we are forced to think about the order in which the marks must be printed, and "such a question about order is usually very illuminating" as you say on page 75. Here is where we consider whether to go to something like LINER or a data structure that would be more appropriate when the marks are very sparse. If the printer has another operation 'new page' where there are 10 lines to a page, or if it has tabulator stops at particular points of a line, etc., all can be decided when we face 'advance to next position' since this is the primary place where we consider the number of operations required (and where we decide what "next position" means). Perhaps the two instructions advance .. . and printsymbol in the above ought to be one since they can't be meaningfully used in another order.

This seems to be further confirmation of your remarks on pages 62 and 63. Decisions about data representation should probably always be deferred until the necessary uses of the data by the algorithm are established.

On pages 63-66 you discuss Wirth's problem of the sequence without repeating blocks. Here I was unable to answer the question "What should the program be like if there is possibly no solution?" I see no good way to do this without go to statements; every way I think of would be better with a go to. This is the difficulty I have in subscribing wholeheartedly to your ideas and perhaps you can show me my error in this example.

On the top of page 67, it seemed to me a better first sketch would be

```
integer s,,t;
s := 1 (and further initialization);
repeat t := s;
       s := "sum of smallest unconsidered decomposition > t"
until s = t;
```

However, both of these "first sketches" fail to indicate all the thinking that goes on. I find it significant that the student spent twenty minutes getting somewhat familiar with the problem, and I venture to say that this was not all wasted; one does not come to the right first sketch until after several other hidden concepts have been discovered.

In fact, let's consider this problem a little. It is desired to find the first repeated element of the multiset $\{a^n + b^n \mid 0 \leq a \leq b\}$. A general way to find duplicates is to sort; and especially since we are looking for the first duplicate, this suggests generating the elements of the set in order. Now the obvious way to generate this set is by a pair of nested loops,

```
for a :=0 step 1 until ∞ do
for b :=a step 1 until ∞ do . . .
```

but this doesn't lead to increasing order. We perhaps think of parallel processes at this point, one for each value of a (since the values for fixed a are increasing; this is the key fact which must be discovered somehow). We imagine a collection of processes generating the values

process[0] : $0^n + 0^n$, $0^n + 1^n$, $0^n + 2^n$, . . .

process[1] : $1^n + 1^n$, $1^n + 2^n$, $1^n + 3^n$, . . .

process[2] : $2^n + 2^n$, $2^n + 3^n$, $2^n + 4^n$, . . .

and we must merge the outputs of these processes into ascending order. We see that at any given time we need not consider process[k+1] until process[k] has gotten its first value used. This sequence of observations seems (to me) to be what underlies the first sketch which magically appears on page 67. But now the first sketch looks rather like this:

```
integer array sum,b[0 : ∞]; comment sum[a] is the next value to be
                            output by process[a], and it corresponds
                            to a^n + b[a]^n;
initiate first process;
repeat find smallest sum among the active processes;
     if it was the first value for that process then initiate the
          next new process;
     advance the process which had smallest sum;
until this sum equals the previously examined sum.
```

I don't believe you get the stated first draft until you have mentally drafted something equivalent to this. By the first declaration in this program I do not mean to commit myself to any particular data structures.

It is interesting to pursue this somewhat further, to the point where we choose appropriate data structures. We soon realize that the (sum,a,b) triples are essentially linked, not independently sum[a],b[a] ; and the proper data structure is a priority queue consisting of these triples ranked on their sum fields. Now we look at Knuth, volume 3 and see which of four or five known methods for priority queues is most appropriate in this instance (probably a sorted list, since the number of processes stays small).

The above program illustrates something else, which I think is important. Whenever I'm trying to write a program without go to statements, I waste an inordinate amount of time in deciding what type of iterative clause to use (while or repeat, etc.). The reason is that our notations aren't really complete. I know in my head what I want to do, but I have to translate it painstakingly into a notation that often isn't well-suited to the mental concept. I know I want to repeat something over and over, and it's easy for me to give a step-by-step description; "first do α , then β , then if y were done, otherwise do δ and we're in the same situation we started." Now this is not suited to present languages since I have to test γ either first or last, writing

$\alpha; \beta;$ while non γ do $\{\delta; \alpha; \beta\};$

or

$\delta^{-1};$ repeat $\delta; \alpha; \beta$ until $\gamma;$

where I invent some trick inverse of δ . (Witness " k := ℓ+1 " on page 71.) Surely you must face the same dilemma. What I really want to say is something like:

loop $\{\alpha; \beta;$ if γ then exit; $\delta\}$ end loop

Since this is a frequent mental construct, in my experience, I believe it deserves a suitable syntax. Otherwise we also find ourselves testing the same condition twice as on page 71 (x = pnt tested three times and one of these is unnecessary).

On page 68 you invite the reader to try writing that silly program himself. I know you haven't time to grade all the readers' solutions, but here is mine anyway. (Unfortunately I did not time myself, I was in bed with a pad of paper, and Jill sleeping beside me, at about 1:00 a.m.; I expect I finished about 15 or 20 minutes later. About 2 minutes were wasted trying to think of a suitable iteration statement.)

```
A begin comment Dijkstra's odd inversion problem:
A char x; comment the character most recently input;
C integer k; char array word[1:20]; comment word[1:k] contains the
C          first k characters of the next word to be printed;
E integer n; the number of words printed so far;
E n := 0;
C k := 0;
A repeat x := RNC;
A        absorb x:
B        if x = sp or x = pnt then
B           print a nonempty word:
C           begin if k > 0 then
C              begin print word:
E                    if n > 0 then PNC(sp);
E                    if odd(n) then
E                       print word backwards:
F                       begin integer i; i := k;
F                          repeat PNC(word[i]);
F                                 i minus 1;
F                          until i < 1;
F                       end
E                    else print word forwards:
G                       begin integer i; i := 1;
G                          repeat PNC(word[i]);
G                                 i plus 1;
G                          until i > k;
G                       end;
E                    n := n+1;
C                    k := 0;
C              end
C           end
B        else add x to word:
D           if k = 20 then
D              word too long error: . . .
D           else begin k := k+1; word[k] := x end;
A until x = pnt;
A PNC(x)
A end.
```

This program has been strung together from the individual pearls A,B,. . .,G which are identified in the left margin. These letters indicate the order in which the decisions were made. I never completed the next step of the development, which would have been pearl H (for the "word too long error"), since I was hurrying and error recoverv is usually not an easy thing. (Perhaps a good solution for that error would be " word[1] := word[20] :=asterisk ".) When I wrote this program I wasn't sure whether or not the first character was required to be a letter, so I allowed for it to be a space.

This program illustrates one thing I wish you would adopt, namely always to give a suitable comment (an 'invariant' essentially) for each declared variable. Programming languages ought to be defined so that such comments are convenient if not mandatory (it's a bother to write the word comment, and a label isn't allowed or appropriate).

Comparing this program to the one devised by your class is interesting, because it is so different. The stated reason for rejecting my form of the outermost loop (bottom of page 68, "the amount of output varies wildly"), is not really to the point; the reason probably was either that (a) they wanted to get started with the meat of the program without stalling around, or (b) it isn't clear what to do with just one character, what does it mean to " absorb x "? The latter problem didn't affect me since I have written so many scanning routines, but admittedly to a novice it will be unclear that a simple finite-state automaton for this input exists. If asked to say what I mean by " absorb x " at level A , however, I would not be able to give a precise definition, other than to say that the program should do what it can to record the fact that it has just read x ; and if a word has just been delimited, it should be output as soon as possible in order to clear out the memory. A precise definition of the absorbtion process is being deferred, for later decisions.

The program found by your students is much more efficient than mine if there are multiple spaces. It isn't easy to patch my program for this, and if I had noticed it I would have had to restructure my program. Curously my main concern while writing that program was not how to pass over spaces quickly, it was when to print a space. I originally had two Boolean variables 'first' and 'even', which I later reduced to the single variable n because odd(n) is a primitive in ALGOL W, and first ≡ (n=0) .

Finally there is the 8 queens problem. On page 76, the remark that 'the only sensible order . . . is the alphabetical order' bothered me a little. For example, Golomb (who discusses precisely this problem in the ACM Journal, 1965, pp. 516 ff.) suggests possibly choosing at each stage the position of x[i] , where i is in the set of unspecified rows, and where x[i] has the least remaining possibilities. (Thus, the position of one queen might already be forced.) Also, Naur suggests starting in the middle since these moves block more later moves.

My main concern though was on page 77 where you give two reasons why a program of the stated structure is less attractive. Your reasons are not convincing, since they would apply with equal force to the program on pages 63-66!

This raises the further question, what would you do if you were asked to produce only one solution (say the alphabetically first one), as in the strings program. Would you reject the recursive program structure just to avoid a "go to" ?

Whoops, I'm afraid I answered my last question; I looked at Wirth's procedure again, and found that he avoids go to by a rather complicated and forced method. Surely a "go to exit" once a solution has been found is conceptually simpler. Please, not all go to's are bad; but it is okay for you to adopt a radical stance on this question in order to heip swing the prevailing balance of opinion the right way.

On page 80, your argument about 28 squares is overstated. Only the squares in unexplored rows need to be updated, so the maximum number of squares to update is 14, 13, 12, 11, 9, 6, 3, 0 for i = 0,1,2,...,7 ; most of the time is spent for $i > 4$, so the average number of updates is less than 10 . This is still-greater than 3 , so the "col, up, down" idea is definitely superior, but the number 28 is much too high.

Finally I suggest a slight improvement in the labeling of the upward diagonals, interchanging up[-i] with up[i] so that the square [n,h] is free if and only if col[h] and up[h-n] and down[n+h] . On many computers this can be tested rapidly for various h by "shift left h, extract" assuming a 38-bit word. Of course, as Golomb remarked, one can find all solutions to the eight queens problem by hand in about an hour, pushing pawns on a chessboard, so there is no need to worry much about efficiency. (After reading Golomb's article I took his suggestion and tried the backtracking method by hand. As I recall, it took me two hours; I missed 3 of the solutions and found one non-solution by mistake. But it is clear that the task requires fairly little computation.)

Cordially,

Donald E. Knuth/pw

Donald E. Knuth
Professor

cc: O.-J. Dahl
R. Floyd
T. Hoare
P. Naur
R. Sites
K. Wirth

DEK/pw

STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

COMPUTER SCIENCE DEPARTMENT

Telephone:
415-321-2300

January 15, 1973

Prof. C. A. R. Hoare
Dept. of Computer Science
The Queen's University of Belfast
Belfast, Northern Ireland

Dear Tony:

You should have recently received a copy of a longish letter I wrote to Dijkstra, about Structured Programming. This is another one, inspired by your chapter. I hope that such discussions of these fundamental issues will prove useful; at least it's good therapy for me, since I like to get my own feelings down on paper.

On the whole, of course, I feel your chapter is magnificent. But there are several points worth debating a little.

1. First, on page 86, lines 13+2, you say "the choice of representation ... must be made as part of the design of the program." Well, the tendency for business data processing these days is to avoid making this decision, by striving for rather abstract programs in which the data representation is self-defining. In other words, large data bases tend to have accumulated over a period of years on various equipment, and the desirable solution is to make each tape (say) begin with a coded description of its own format; the programs should dynamically accommodate each format. We may someday therefore see computers which run abstract programs. (The G-20 and B6700 are already something like this.)

2. Your discussion of the concept of type seems to omit the idea of subtypes (something like SIMULA subclasses). For example, if p and q are prime numbers, they are also integers so they inherit all the axioms of integers. (I don't understand your remark about "distinctions of an arbitrary kind" on page 91, line 26; furthermore, mathematicians most frequently use the letters p and q , sometimes l , for primes.)

   This reminds me of the very interesting language AUTOMATH, invented by Dijkstra's colleague (and next-door neighbor) N. G. de Bruijn. AUTOMATH is not a programming language, it is a language for expressing proofs of mathematical theorems. The interesting thing is that AUTOMATH works entirely by type declarations, without any need for traditional logic! I urge you to spend a couple of days looking at AUTOMATH, since it is the epitome of the concept of type.

When I last looked at AUTOMATH it did not contain the concept of subtypes, and my impression was that many proofs in AUTOMATH would be shorter by an order of magnitude if subtypes were allowed; however, it would complicate the language (and the compiler/proof-checker) to an indeterminate extent. Perhaps you and I can' look into this further next year at Stanford.

3. On page 93, you state that "Arbitrary real numbers . . . can be represented by . . . program structures." Of course you mean only the computable real numbers!

4. At the bottom of page 94, and again on page 99, paragraph (4), you make a statement that sounds reasonable at first and which many language designers have been following . . . but on further examination it appears to be wrong. The statement is, more or less, that some types ought to be unordered since their relative order is meaningless to a programmer.

   I recently came upon an interesting example which seems to refute this postulate; or at any rate there was no way in SIMULA that I could write an efficient program, the language forced me to be inefficient! Here was the application: I had a data type

   type reflist = sparse powerset of ref(object)

   and I wanted to represent it as a list of references. Given two such reflists , of sizes m and n, my algorithm needed to test whether they had any common elements. Obviously this would take about m+n steps if I could keep the reflists ordered, but SIMULA allows only equal-unequal comparison of references. Therefore I was forced to use an algorithm which required mn steps!

   Here is a case where the ordering of reference variables has no semantic meaning, yet my program would work meaningfully (and much faster) if I allowed the machine to order the reference variables in any arbitrary but consistent way. Traditional garbage collection and compaction algorithms, at least the in-core versions, preserve this arbitrary relative order even when they reallocate memory.

5. My impression on page 100 is that you are hanging too much on the concept of ordered type. By your definitions, an unordered type must have arbitrary sequencing while an ordered type must have min-to-max sequencing. It seems better to me to separate the concepts of order and sequencing, by having various sequencing operations; an ordered type could still be scanned in arbitrary order in an abstract program if the programmer says so, because he will prove the correctness for an arbitrary order (and he will therefore know that he has additional freedom in his later choice of concrete representations).

6. On the top of page 103, the two procedures called "deal with single character" cannot both be the same, because "buffer" is the character to be dealt with only in the second case. You might change the first line to "else deal with single colon character".

7. Your program on page 107 allows the invalid date Feb 29, 1900. (Perhaps you could simply restrict type year to 1901...1969.) Incidentally, I wonder what EWD would do about the goto's in this program?

8. When we get to "discriminated unions" I begin to wonder about your choice of notations, since you seem on the one hand to be trying to minimize the character set (the comma and semicolon and colon are used in several different senses, and "in"is used for ϵ , etc.), while on the other hand you make use of $\wedge$ and $\vee$ and even $\cap$, which are very rare in computer hardware. The notation for discriminated union seems especially wrong to me; that comma isn't a weak enough delimiter. Conventionally in English, comma is a shorter break (i.e., stronger in precedence) than semicolon, and semicolon is a shorter break than a colon. This order has already been violated (inverting : and ;), but that isn't really bad; the trouble is that the comma has already got a precedence stronger than either of these and your language should be self-consistent. For example, wouldn't it be natural for a programmer to abbreviate your example on page 111 to

type patience card = (red,blue:cardface)

before realizing this means something else? I would recommend using another symbol for discriminated union, preferably the " | " from BNF. Note that this would look especially nice in your parsing example.

9. On page 114, are those tag fields and compile-time case discriminations advisable even when the program has been proved correct?

10 On page 116, lines 10 and 11, I'm amazed at your curiously restrictive use of the word "table". What about a table of prime numbers less than 100 , etc.?

11 I was also surprised on page 123 that you didn't discuss the similarities and differences of

type T = powerset T'

and type T = array T' of Boolean

12. Page 125, top, I find these notations unfortunate, especially x :-y which conflicts with SIMULA conventions. By analogy, wouldn't x :=y now have to mean that x is replaced by x=y (say when x and y are Boolean variables)? The colon is being overused again.

Of course I must think of a better alternative. Dijkstra's paper used " i plus 1 " on pp. 54-60, but i := i+1 elsewhere. ALGOL 68 has i :+= 1 . None of these really satisfies. What we seem to need is some "reflexive" symbol (like the German "sich" referring to the subject). Denoting this unknown symbol by ❑ , we want to have x ❑ op y be equivalent to x := x op y , for all variables x and all operators op . Maybe ❑ could be *= , read "self-replaced"?

13 Page 129, "if next.w > W then exit primefinder": Yes, yes, bravo!

14 Page 132. Actually cars are verboten as examples ever since LISP was invented.

15 Page 146, bottom. I don't understand what you mean, "the axiom of exclusion". Is it von Neumann's "axiom of regularity"?

16. Your example of examination timetables is beautiful, but I wish it had been carried off with a bit more finesse.

First in the definition of "suitable" on page 160, there is no need to say "-trial" in the assignment to untried , since trial = {e} and e has already been removed. (This is fortunate, because you later decide to represent trial as a sequence and the other operands as bitstrings.) But the big awkwardness occurs in gensupersets; the introduction of save 1 and save 2 is not clever nor is it art! In the first place there is no need to say that gensupersets preserves "untried", since the value of the latter is never used after gensupersets. This eliminates save 1 . Secondly, the purpose of save 2 is to restore untried at another place, and there is no need for the trickery you pulled; instead, " save 2 := untried; untried :- incompat(e) " and later " untried :=save 2 " would be shorter (and faster in your eventual representation). But in fact it somehow is clear that untried shouldn't be a global variable that is explicitly saved and restored, it is a natural parameter to gensupersets. Thus, the entire program on page 161 becomes much simpler and cleaner:

```
procedure gensupersets(untried: powerset exam);
begin e: exam;
   record;
   if size(trial) < k then
      while untried ≠ { } do
         begin e from untried;
            trial: v{e};
            if sessioncount(trial) < hallsize then
                  gensupersets(untried - incompat(e));
            trial:- {e}
         end;
end gensupersets.
```

It would perhaps be interesting to analyze what made you go wrong here, and to "abstract" the source of the error, since presumably it is something students need to be taught to avoid.

Another point is that you haven't declared the procedure "sessioncount". Since it appears in the innermost loop it is clear that actually the sessions should be redeclared as

type session = {exams: powerset exam; sessioncount, size: integer].

This is a very important consideration in this algorithm, so I was sorry to see it neglected.

Still another point is the representation of timetable on page 164; this is evidently an output variable (as you define on page 135), except that you consider there only the case of sequences not powersets. The best representation for timetable in your example is to print it as you go.

And there is yet another point to make. The first representation of exam that comes to mind is not necessarily the integer subrange 0...500 , really a "sparse powerset sequence character" is more natural at least in the external real world representation. Expecting 500 courses to be assigned a unique integer code number between 0 and 500 is quite impossible in the real world. So here we have another interesting (and typical) situation: the same type (exam) wants two different representations in different parts of the algorithm, and we must convert between them at the interface.

Please excuse my gloating over all these improvements. It is much easier for me to improve your program than for you to have composed it in the first place; I'm just a Monday-morning quarterback. The point is that this timetable example is a vehicle for illustrating even more things than you expected.

17. Since you are editor of this outstanding series of books for Academic Press, I think you ought to give some thought to the standards for typesetting, especially of ALGOL programs. About 40-50 years ago, G. H. Hardy made a study of mathematical composition, for Oxford University Press, and the resulting standards have been widely adopted. (A short and fascinating booklet explaining them has been published: The Printing of Mathematics, by Chaundry et al., Oxford U. Press, 1954. I recommend it!) One of these sacred rules is to insert small spaces around every equals sign; and unfortunately Academic Press hasn't been told not to do this in ALGOL! A proper letter from you will cause them to set " x := y " instead of " x: = y ". (My spacing here is exaggerated, but I know they can do better than they have done on the spacing.) The same should apply to your x :∨ y and so on, if you still want to stick to these. The second thing you ought to consider carefully is the use of italics. At present they are setting one-character variable names in italic type,

multi-character names in Roman type, This doesn't look so pleasing to me; see especially pages 112 " cardl.normal.r ", and page 128 at the top. The ACM conventions for ALGOL (which I think have been written down by Myrtle Kellington, you could write to her) are more to my taste. Have you noticed that, ever since the ALGOL report was originally published in 1960, there is an interesting typographical distinction between italics and Roman letters (besides the obvious distinction with boldface letters)? Perhaps Peter Naur originally suggested this. In the syntax for basic symbols, all the letters are italics; and all identifiers are consistently printed in italics, whether they are one-character or multi-character. On the other hand, ALGOL 60 allows any set of characters (including summation signs, etc.) to appear in comments and strings; and as if to prove this, they traditionally use Roman letters in comments and strings, except when an identifier of the program is mentioned.

18. This letter is long enough. Thank you again for teaching me a lot by writing your monograph.

Sincerely,

Donald E. Knuth /pm

Donald E. Knuth
Professor

cc: O.J. Dahl
E. W. Dijkstra
R. W. Floyd
P. Naur
R. L. Sites
N. Wirth

P.S. Here are some comments on your paper "Proof of Correctness of Data Representation" in Acta Informatica (December 1972).

On page 273, I suspect the original form of the procedure "has" was called "contains" because it doesn't end has ! In this example, the second for loop in "remove" can be replaced by simply " A[j] := A[m] ". Did you consciously avoid this for some reason?

On page 277, I don't see why the proof of "has" says merely " $j < m$ " while the proof of "insert" says " $0 \leq j \leq m$ ". Surely the lower bound on j is needed in both places since $A[j+1]$ is used. But the most curious thing is that the condition $m \leq 100$ is a necessary premise, but it is not shown. Thus the lemma for "has" should begin " $m \leq 100$ & $0 \leq j < m$ & $j < m$ & . . . ".

Finally, I enjoyed the closing acknowledgment since you yourself are the author of two of the referenced works!

P.P.S. Typos and minor corrections:

p. 86, line 11, coefficients

p. 91, line 20, "Let S be a family of sets of integers"

p. 110, lines 28-29, local: local car,
foreign: visitor car

p. 113, lines 3-7, change $s_1, s_2, \cdots, a_n$ to $S_1, S_2, \ldots, S_n$.

p. 132, line 33, type deck = sequence cardface;

p. 156, line 26, (s1 $\wedge$ s2 = { }) or (s1 = s2)

p. 160, line 11, e from remaining;

p. 162, lines 15-16, "remainder" should be "remaining"

p. 162, line -4, untried := save 2;

p. 164, line 21, operations on a session

p. 166, line -2, theorems (?)

p. 171, line 15, in

p. 172, x :-y is not defined

p. 173, line -3, $d_1, \ldots, d_n$ for $x_1, \ldots, x_n$.

DEK/pw

STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

COMPUTER SCIENCE DEPARTMENT

Telephone:
415-321-2300

April 12, 1973

Prof. Ole-Johan Dahl
Matematisk Institutt
Universitetet i Oslo
Blindern, Oslo 3, Norway

Dear O-J:

This is the third and last in the series of letters to myself based on the book Structured Programming. Your chapter is certainly a masterful conclusion to this important book. It sets forth the key virtues of SIMULA in an especially clear and compelling fashion.

My comments aren't deep but you may be interested in a few reflections I had as I read the chapter.

1. Page 183. The histogram example does not completely remove the artificial separation of the operational and data storage aspects, because the array limits used to initialize the histogram must be retained (and never changed) during the program execution. The programmer must be aware of this connection, he must treat "real array A[1:7], B[1:12]; . . . initialise A,B..." at a conceptual level next to the histogram class and not at the conceptual level of the rest of the program. He must be warned that the use of A extends after the use of "new histogram (A,7)", assuming that he hasn't read the code for the histogram class. This may seem a minor point, but somehow I don't think it is completely negligible; it demonstrates a conceptual need for read-only variables.

2. The word "detach" had always seemed to me machine-oriented instead of problem-oriented, and it sounded quite mysterious. Your explanation here has cleared it up for me, for the first time. It is like the word "return" except at a more global level. I suppose a concept of superdetaching and supercalling might exist, at a still more global level, though I don't see any important applications.

3. Page 193. I tried the suggested permutation procedure "based on the same swapping strategy, which returns with the numbers in reverse order", and it didn't work; at least, the swapping strategy has to be changed. Otherwise we have

original state:	1 2 3 4 5
after swap(p[1],p[5]):	5 3 2 1 4
after swap(p[2],p[5]):	1 4 5 5 2
after swap(p[3],p[5]):	5 3 2 1 4

I can't see anything better than e.g. swapping with p[1], p[k-1], p[3], p[k-3], ... , and ending with a different transformation depending on whether k is even or odd.

4. This permuter class does not rely on the fact that the numbers permuted are the integers 1 to n nor that they are initially in order. Therefore it seems slightly better to have p as a parameter:

```
class permuter (p,n); integer array p; integer n;
```

(Unfortunately Algol makes us commit ourselves to integer arrays.)

An amusing and quite natural way to write the declaration, using coroutines instead of procedures, now presents itself-

```
begin Boolean more;
    more := true;
    if n = 1 then detach
    else begin ref (permuter) r; integer i,q;
        for i := 1 step 1 until n do
        begin r :- new permuter (p,n-1);
            while r.more do
                begin detach; call (r) end;
            if i < n then
                begin:= p[i]; p[i] := p[n]; p[n] := q;
                    detach;
                end
        end;
        q := p[1]; for i := 1 step 1 until n-1 do p[i] := p[i+1];
        p[n] := q;
    end;
    more := false;
end of per-muter.
```

5. Another permuter algorithm can be based on Trotter's algorithm. This is interesting because (a) it's faster [n-1 times out of n the operation is quite simple'; (b) it uses a class defined within another class; (c) the program in these terms displays the idea behind Trotter's method while there is no way to do this using Algol's recursive procedure control.

```
class Tpermuter(p,n); integer n; comment assume n > 2;
   begin Boolean more; more := true;
         integer t; comment the current "offset";
      begin s s permute(k); integer k;
         begin if k = 2 then
            begin detach; swap(p[t],p[t+1]);
                  detach; swap(p[t],p[t+1]);
                  more := false; detach
            end else
            begin ref (permute) r; integer i;
               r :- new permute(k-1); detach;
               while more do
                  begin for i := 1 step 1 until k-1 do
                     begin swap(p[i+t-1],p[i+t]); detach end;
                     call(r); detach;
                     for i := k-1 step -1 until 1 do
                        begin swap(p[i+t-1],p[i+t]); detach end;
                           t := t+1; call(r); detach;
                  end while more;
            end k > 2 case
         end of permute;
         ref (permute) r; integer t;
         r :- new permute (n);
         while more do
            begin t := 1; call(r); detach end;
      end
   end of Tpermuter
```

The idea is to swap(p[1],p[2]); swap(p[2],p[3]);. . . ; swap(p[n-1],p[n]); then do the next step for sequence n-1 ; then swap(p[n-1],p[n]); . . . ; swap(p[1],p[2]); then do the next step for n-1 but shifted right one; then start *over* again.

6. The syntax example you give is very beautiful, of course; I think you should credit it to Bob Floyd, who was the first to publish it (IEEE Trans. on Elect. Computers 1964, the same issue as my article on SOL). He presented it in terms of men in a corporation (almost like Chaplin's officers!), and at this time Bob and I corresponded about how to express the algorithm properly using "recursive coroutines" since it was clear that recursive subroutines were insufficient. This example is what first taught me about the limitations of ALGOL's recursion.

   As I recall, Bob and I expressed the algorithm somewhat less elegantly at that time; we had one "class" declaration for every syntactic type, and the programs were complicated by using only resume/resume sequencing, so that every object had to know the name of its superior. We wanted a symmetrical way to pass information between coroutines, and I think we used local variables instead of global variables for this purpose. So you can see why I was so pleased to see SIMULA when you first sent me its description in 1965!

7. I don't understand why you call the shortest-path algorithm the Lee algorithm, when it is generally credited to Dijkstra [Numerische Mathematik 1 (1959), 269-271]. I don't know what Lee you refer to; but if he came after Dijkstra, the correct reference should appear in a book co-authored by Dijkstra!

8. Your discussion of programming levels by means of prefixed blocks is very thought-provoking. (This is partly why I missed the idea of subtypes in Chapter 2.) As I was reading Section 7 it finally dawned on me that this may be the way to string together the "pearls" which Dijkstra described in Chapter 1, making each independent design decision correspond to a prefix. Therefore I went back to the program in my letter to Dijkstra, for that word-reversal problem, where the pearls were identified by letters in the left margin. I wrote the following code top-down almost exactly as it appears here (therefore writing " B class A " before having any idea what B would involve, only knowing that it represents a lower level which will be specified later!):

```
B class A; begin comment Dijkstra's odd inversion problem;
    char x; comment the character most recently input;
    repeat x := RNC;
           absorb(x);
    until x = pnt;
    PNC (x)
  end A;
C class B; begin comment explain what "absorbing" means;
    procedure absorb(x); char x;
    if x = sp or x = pnt then print a nonempty word
    else add to word(x);
  end B;
E class C; begin comment facilities for word memory*,
    integer k; char array word[1:20]; comment word[1:k] contains
       the first k characters of the next word to be printed;
    procedure print a nonempty word;
       if k > 0 then begin print word; k := 0 end;
    procedure add to word(x); char x;
       if k = 20 then word too long error
       else begin k := k+1; word[k] := x end;
    k := 0;
  end C;
F class E; begin comment handle the even-odd requirement and spacing;
    integer n; comment the number of words printed so far;
    Procedure print word;
       begin if n > 0 then PNC(sp);
          if odd(n) then print word backwards else print word forwards;
          n := n+1
       end;
    n := 0;
end E;
```

```
If class F; begin comment the way to print a word;
      virtual integer k; virtual char array word[1:20]; comment, word[1:k
         contains the first k characters of the next word to be printed;
      integer i;
      procedure print word backwards;
         begin i := k;
            repeat PNC(word[i]); i minus 1;
            until i = 0;
         end;
      procedure print word forwards;
         begin i := 1;
            repeat PNC(word[i]); i plus 1;
            until i >k;
         end;
   end F;
```

The next lower level, H , will define the 'word too long error' procedure (curiously this seems to be a high level operation but it appears lower), and the next level will-perhaps define PNC and RNC, etc.

Comparing this program with the earlier one leads to the following observations:

(a) All the "pearls" now do appear in one place. (I combined C and D , also F and G, as being essentially on the same level.) Furthermore each class does seem to make sense as a fairly isolated conceptual unit.

(b) This program, if executed, would involve considerable procedure-call overhead. A smart compiler will remove it. Some languages have a way to specify that a procedure is 'in-line', meaning that it is to be explicitly expanded wherever it is called.

(c) In level F , I needed some virtual declarations. When writing programs in this top-down style I suspect that programmers will often slip up and forget to include the necessary virtual declarations. An alternative would be to pass (k, word) as parameters through the various levels; but that doesn't seem natural to me somehow, perhaps it should.

(d) I just noticed that my comment explaining k and word is not sufficient to prove that the program works; somehow it must be stated that k is large enough to contain as much of the word as has been absorbed so far. This points out a common difficulty with program proving: We can prove the validity of algorithms that solve familiar mathematical problems, but when it comes to real software problems it is often hard to state the invariants because they involve concepts for which no standard notation exists.

9. Finally, a minor point on page 219. You wouldn't want to have the first order generate a reporter object as stated; for it would mean inserting the " dt " specification at an extremely awkward place in the input (within the first order's description). Better would be to have reporter generate the first order, just before its "while true", and to change line 25 of the program to
simulate(new reporter(inreal), lim)

To sum up all these letters of mine, I can't remember ever having read a computer science book that was so thoroughly stimulating from cover to cover, and I want to thank all three authors again for the considerable effort that has gone into it. This book will certainly have a profound impact on the future of computer science.

Sincerely,

Donald E. Knuth/pw

Donald E. Knuth
Professor

P.S. Now that I've written all these letters, I wonder if it might not be useful to circulate them to a wider audience by issuing them as a Stanford C.S. report. (Not to be published in a journal, but to go out to say 400 readers in typewritten form.) Please tell me if you think this is a bad idea. I think it might contribute to the discussion of your book in various circles, since structured programming is "making so many waves" these days.

P.P.S. Here also are some typographical errors I noticed; many of them are technically nontrivial, so you should check me:

page 176, line 11, "As the . .."
page 182, class histogram + 5, should say Y > X[i+1]
page 182, line -5, "called by reference.++
page 187, illustration, "detach" not "detatch"
page 193, lines -5 and -6, change "tree" to "binary tree"
page 197, syntax rule (5), mult sign not " X "
page 202, line before section 6, delete " . " after "so on"
page 203, line -11, delete " 1 " after "part"
page 206, line -3, delete "should"
page 207, line 8, add " ; " after "list"
page 207, line 9, change "should be" to "are"
page 207, line -11, change "linkage-list" to "list"
page 207, line -2, " y " is wrong font
page 208, line 1, " L " is wrong font
page 213, line 1, put " ; " at end of line
page 216, line -10, quotes around "no ...limit"
page 217, line -13, and four places on the next two pages, "mgroup" should not be partly italics!
page 220, reference (8), "Action".
page 220, reference (10), "pp. 401-414."
page 219, line 9, delete " v "

Since I foresee this book being reprinted often, I imagine this list of misprints will be useful.

DEK/pw

cc : Prof. Dr. E. W. Dijkstra
Prof. R. W. Floyd .
Prof. C. A. R. Hoare
Prof. P. Naur
Prof. N. Wirth