On the Composition of Well-Structured Programs†

NIKLAUS WIRTH

Federal Institute of Technology (ETH) Zurich, Switzerland

A professional programmer's know-how used to consist of the mastery of a set of *techniques* applicable to specific problems and to some specific computer. With the increase of computer power, the programmers' tasks grew more complex, and hence the need for a systematic approach became evident. Recently, the subject of programming *methods*, generally applicable rules and patterns of development, received considerable attention. "*Structured programming*" is the formulation of programs as hierarchical, nested structures of statements and objects of computation. We give brief examples of structured programs, show the essence of this approach, discuss its relationship with program verification, and comment on the role of structured languages.

Keywords and phrases: programming methods, systematic programming, program schemas, GOTO-free programs, well-structured programs, Pascal.

CR categories: 1.50, 2.43, 4.0, 4.20, 5.24, 5.25

INTRODUCTION

In the first decade of computers, say up to the early sixties, computers were quite limited in their power. The task of the programmer was to formulate algorithms in the specific order codes of these machines so that they were utilized as effectively as possible. Primarily because of their limitations, this task was achieved by collecting sets of clever techniques and startling tricks, and by finding applications for them as frequently as possible. Examples of such techniques were the programmed self-modification of parts of the program, such as, for instance, the conversion of conditional jumps into dummy instructions and vice versa, or the sharing of store for functionally independent, but never simultaneously used auxiliary variables.

Tricks were necessary at this time, simply because machines were built with limitations imposed by a technology in its early development stage, and because even problems that would be termed "simple" nowadays could not be handled in a straightforward way. It was the programmers' very task to push computers to their limits by whatever means available. We should recall that the absence of index registers (and indirect addressing), for example, made automatic code modification a mere necessity (see also [1] and [2]).

The essence of programming was understood to be the *optimization of the efficiency* of particular machines executing particular algorithms. As computers grew more powerful, the problems posed to the programmers grew proportionally, and as a result, the growing power of hardware did not ease, but rather increased the burden. The elimination of deficiencies, errors and blunders—called debugging—became the overwhelming problem.

† This article is a revised and extended version of a presentation made at International Computing Symposium 1973, Davos, Switzerland, September 1973.

CONTENTS

Understandably, the remedy was sought in the development and use of better tools in the form of programming languages. The amount of resistance and prejudices which the farsighted originators of FORTRAN had to overcome to gain acceptance of their product is a memorable indication of the degree to which programmers were preoccupied with efficiency, and to which trickology had already become an addiction. However, once these adversities and fears had been overcome, FORTRAN had a tremendous impact—an impact that is still felt today. ALGOL 60 followed several years later; it went beyond FORTRAN in several significant respects, but essentially shared the same purpose and intention. In particular, it extended to the level of statements what FORTRAN had introduced on the level of (arithmetic) expressions: *structure*. But ALGOL 60 was not very successful when measured by its frequency of use in technical and commercial applications. There are many reasons for this, one being that it appeared on the scene when the relevance of structure had not yet been widely recognized, and its restrictiveness against the use of clever tricks was considered to be a handicap and a deficiency. The law of the "Wild West of Programming" was still held in too high esteem! The same inertia that kept many assembly code programmers from advancing to use FORTRAN is now the principal obstacle against moving from a "FORTRAN style" to a structured style.

As the power of computers on the one side, and the complexity and size of the programmer's task on the other continued to grow with a speed unmatched by any other technological venture, it was gradually recognized that the true challenge does not consist in pushing computers to their limits by saving bits and microseconds, but in being capable of organizing large and complex programs, and assuring that they specify a process that for all admitted inputs produces the desired results. In short, it became clear that any amount of efficiency is worthless if we cannot provide *reliability* [4]. But how can this reliability be provided? Here structure enters the scene as the one essential tool for mastering com-

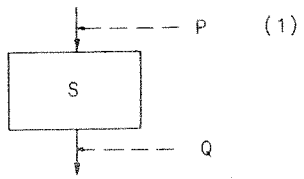
plexity, the effective means of converting a seemingly senseless mass of bits or characters into meaningful and intelligible information. We must recognize the strong and undeniable influence that our language exerts on our ways of thinking, and in fact defines and delimits the abstract space in which we can formulate—give form to—our thoughts.

But now the term *structured programming* has been coined, and it seems finally to be achieving what the term "structured language" was unable to suggest. It was first used by E. W. Dijkstra [3], and has spread with various interpretations and connotations since then. It is the expression of a conviction that the programmers' knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically, and that particular techniques should be replaced (or augmented) by a method. At its heart lies an *attitude* rather than a recipe: the admission of the limitations of our minds. The recognition of these limitations can be used to our advantage, if we carefully restrict ourselves to writing programs which we can manage intellectually, where we fully understand the totality of their implications.

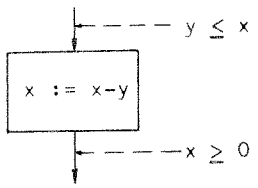
1. INTELLECTUAL MANAGEABILITY OF PROGRAMS

Our most important mental tool for coping with complexity is *abstraction*. Therefore, a complex problem should not be regarded immediately in terms of computer instructions, bits, and "logical words," but rather in terms and entities natural to the problem itself, abstracted in some suitable sense. In this process, an abstract program emerges, performing specific operations on abstract data, and formulated in some suitable notation—quite possibly natural language. The operations are then considered as the constituents of the program which are further subjected to decomposition to the next "lower" level of abstraction. This process of *refinement* continues until a level is reached that can be understood by a computer, be it a high-level programming language, FORTRAN, or some machine code [5, 6].

For the intellectual manageability, it is crucial that the constituent operations at each level of abstraction are connected according to sufficiently simple, well understood *program schemas*, and that each operation is described as a piece of program with *one starting point* and a *single terminating point*. This allows defining states of the computation (P, Q), i.e., relations among the involved variables, and attaching them to the starting and terminating points of each operation (S). It is immaterial, at this point, whether these states are defined by rigorous mathematical formulas (i.e., by predicates of logical calculus) or by sufficiently clear and informative sentences, or by a combination of both. The important point is that the programmer has the means to gain clarity about the interface conditions between the individual building blocks out of which he composes his program [7].



Example



An example may clarify the issues at this point. The reader should be aware that any example that is sufficiently short to fit onto a single page cannot be much more than a metaphor, probably unconvincing to habitual skeptics. The important thing is to abstract from the example and to imagine the same method being applied to large programming problems.

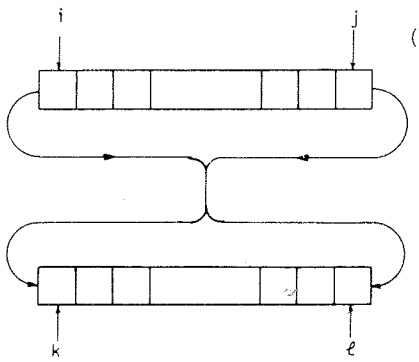
Example 1: Sequential Merging

Given a set of $n = 2^N$ integer variables $a_1 \cdots a_n$, find a recipe to permute their values such that $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$ using

the principle of sequential merging. Thus, we are to sort under the assumption of strictly sequential access. Briefly told, we shall use the following algorithm:

1) Pick individual components $a_i^{(1)}$ and merge them into ordered pairs, depositing them in a variable $a^{(2)}$.
2) Pick the ordered pairs from $a^{(2)}$ and merge them pairwise into ordered quadruples, depositing them in a variable $a^{(3)}$.
3) Continue this game, each time doubling the size of the merged subsequences, until a single sequence of length $n = 2^N$ is generated.

At the outset, we notice that two variables $a^{(1)}$ and $a^{(2)}$ suffice, if the items are alternately shuttled between them. We shall introduce a single array variable A with $2n$ components, such that $a^{(1)}$ is represented by $A[1] \cdots A[n]$ and $a^{(2)}$ is represented by $A[n+1] \cdots A[2n]$. Each of these two conceptually independent parts has two points of sequential access, or read/write heads. These are to be denoted by pairs of index variables i, j and k, l respectively. We may now visualize the sort process as a repeated transfer under merging of tuples *up* and *down* the array A.



(2)

The first version of our program is evidently a repetition of the merge shuttle of p-tuples, where each time around p is doubled and the direction of the shuttle is changed. As a consequence, we need two variables, one to denote the tuple size, one to denote the direction. We will call them p and *up*. Note that each repetitive operation must contain a change of its (control) variables within the loop, an initialization in front of the loop, and a termination condition. We easily convince ourselves of the correctness of the following program:

```
up := true; p := 1;                                   (3)
repeat 1: "initialize indices i, j, k, and
           l";
       2: "merge p-tuples from i- and
           j-sequences into k- and
           l-sequences";
          up := ¬up; p := 2*p
until p = n
```

Statement-1 is easily expressed in terms of simple assignments depending on the direction of the merge pass:

```
1: if up then                                         (4)
     begin i := 1; j := n;
           k := n+1; l := 2*n
     end
   else
     begin k := 1; l := n;
           i := n+1; j := 2*n
     end
```

Statement-2 describes the repeated merging of p-tuples; we shall control the repetition by counting the number m of items merged. The sources are designated by the indices i and j; the destination alternates between indices k and l. Instead of introducing a new variable standing alternately for k and l, we use the simple solution of interchanging k and l after each p-tuple merge, and letting k denote the destination index at all times. Clearly, the increment of k has then to alternate between the values $+1$ and -1; to denote the increment, we introduce the auxiliary variable h. We can easily convince ourselves that the following refinement is correct:

```
2: begin m := n; h := 1;                              (5)
     repeat m := m−2*p;
       3: "merge one p-tuple from
           each of i and j to k, in-
           crement k after each
           move by h"; h := −h;
       4: "exchange k and l"
     until m = 0
   end
```

Whereas statement-4 is easily expressed as a sequence of simple assignments, statement-3 involves more careful planning. It describes the actual merge operation, i.e., the repeated comparison of the two incoming items, the selection of the lesser one, and the stepping up of the corresponding index. In order to keep track of the number of items taken from the two sources, we introduce the two counter variables q and r. It must be noted that the merge always exhausts only one of the two sources, and leaves the other one nonempty. Therefore, the leftover tail must subsequently be copied onto the output sequence. These deliberations quickly lead to the following description of statement-3:

```
3: begin q := p; r := p;                                   (6)
     repeat {select the smaller item}
       if A[i] < A[j] then
         begin A[k] := A[i];
               k := k+h; i := i+1;
               q := q-1
         end
       else
         begin A[k] := A[j];
               k := k+h; j := j-1;
               r := r-1
         end
     until (q = 0) ∨ (r = 0);
     5: "copy tail of i-sequence";
     6: "copy tail of j-sequence"
   end
```

The manner in which the tail copying operations are stated demands that they be designed to have no effect, if initially their counter is zero. Use of a repetitive construct testing for termination *before* the first execution of the controled statement is therefore mandatory.

```
5: while q ≠ 0 do                                          (7)
     begin A[k] := A[i];
           k := k+h;
           i := i+1; q := q-1
     end

6: while r ≠ 0 do
     begin A[k] := A[j];
           k := k+h;
           j := j-1; r := r-1
     end
```

This concludes the development and presentation of this program, if a computer is available to accept statements of this form, i.e., if a suitable compiler is available.

In passing, I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, "top-down" manner. Later refinement steps may often show that earlier decisions are inappropriate and must be reconsidered. But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness. The emerging modularity is particularly welcome if programs have to be adjusted to changed or extended specifications. This is a most essential advantage, since in practice few programs remain constant for a long time. The reader is urged to rediscover this advantage by generalizing this merge-sort program by allowing n to be any integer greater than 1.

Example 2: Squares and Palindromes

List all integers between 1 and N whose squares have a decimal representation which is a palindrome. (A palindrome is a sequence of characters that reads the same from both ends.)

The problem consists in finding sequences of digits that satisfy two conditions: they must be palindromes, and they must represent squares. Consequently, there are two ways to proceed: either generate all palindromes (with $\log N^2$ digits) and select those which represent squares, or generate all squares and then select those whose representations are palindromes. We shall pursue the second method, because squares are simpler to generate (with conventional programming facilities), and because for a given N there are fewer squares than palindromes. The first program draft then consists of essentially a single repetitive statement.

```
n := 0;                                                    (8)
repeat n := n + 1; generate square;
```

```
    if decimal representation of square
        is a palindrome
      then write n
until n = N
```

The next step is the decomposition of the complicated, verbally described statements into simpler parts. Obviously, before testing for the palindrome property, the decimal representation of the square must have been computed. As an interface between the individual parts we introduce auxiliary variables. They represent the result of one step and function as the argument of the successive step.

$d[1] \cdots d[L]$	an array of decimal digits
L	the number of digits computed
p	a Boolean variable

(note that $L = entier(2\ log\ N) + 1$)
The refined version of (8) becomes

```
n := 0;                                          (9)
repeat n := n+1; s := n*n;
  d := decimal representation of s;
  p := d is a palindrome;
  if p then write (n)
until n = N
```

and we can proceed to specify the three component statements in even greater detail. The computation of a decimal representation is naturally formulated as the repeated computation of individual digits starting "at the right".

```
L := 0;                                          (10)
repeat L := L+1;
  separate the rightmost digit of s,
    call it d[L]
until s = 0
```

The separation of the least significant digit is now easily expressed in terms of elementary arithmetic operations as shown in (12). Hence, the next task is the decomposition of the computation of the palindrome property p of d. It is plain that it also consists of the repeated, sequential comparison of corresponding digits. We start by picking the first and the last digits, and then proceed inwards. Let i and j be the indices of the compared digits.

```
i := 1; j := L;                                  (11)
repeat compare the digits;
  i := i+1; j := j-1
until (i ≥ j) or digits are unequal
```

A last refinement leads to a complete solution entirely expressed in terms of a conventional programming language with adequate structuring facilities.

```
n := 0;                                          (12)
repeat n := n+1; s := n*n; L := 0;
  repeat L := L+1;
    r := s div 10; d[L] := s - 10*r;
    s := r
  until s = 0;
  i := 1; j := L;
  repeat p := d[i] = d[j];
    i := i+1; j := j-1
  until (i ≥ j) or ¬p;
  if p then write (n)
until n = N
```

This ends the presentation of Example 2.

2. SIMPLICITY OF COMPOSITION SCHEMES

In order to achieve intellectual manageability, the elementary composition schemes must be simple. We have encountered most of the truly fundamental ones in this second example. They encompass *sequencing*, *conditioning*, and *repetition* of constituent statements. I should like to elaborate on what is meant by simplicity of composition scheme. To this end, let us select as example the repetitive scheme expressed as

```
while B do S                                     (13)
```

It specifies the repeated execution of the constituent statement S, while—at the outset of each repetition—condition B is satisfied. The simplicity consists in the ease with which we can infer properties about the **while** statement from known properties of the constituent statement. In particular, assume that we know that S leaves a property P on its variables unchanged or *invariant* whenever B is true initially; this may be expressed formally as

$$P \wedge B\ \{S\}\ P \qquad (14a)$$

according to the notation introduced by Hoare [8]. Then we may infer that the **while** statement also leaves P invariant, regardless of the number of times S was repeated. Since the repetition process terminates only after condition B has become false, we may infer that in addition to P, also $\neg B$ holds after the execution of the **while** statement. This inference may be expressed formally as

$$P\ \{\textbf{while}\ B\ \textbf{do}\ S\}\ P \wedge \neg B \quad (14b)$$

This formula contains the essence of the entire **while**-construct. It teaches us to look for an invariant property P, and to consider the result of the repetition to be the logical combination of P and the negation of the continuation condition B. A similar pattern of inference governs the **repeat**-construct used in the preceding examples. Assuming that we can prove

$$Q \vee (P \wedge \neg B)\{S\}P \quad (15a)$$

about S, then we may conclude that

$$Q\ \{\textbf{repeat}\ S\ \textbf{until}\ B\}\ P \wedge B \quad (15b)$$

holds for the **repeat**-construct.

There remains the question, whether all programs can be expressed in terms of hierarchical nestings of the few elementary composition schemes mentioned. Although in principle this is possible, the question is rather, whether they can be expressed conveniently, and whether they *should* be expressed in such a manner. The answer must necessarily be subjective, a matter of taste, but I tend to answer affirmatively. At least an attempt should be made to stick to elementary schemes before using more elaborate ones. Yet, the temptation to rescind this rule is real, and the chance to succumb is particularly great in languages offering a facility like the **goto** statement, which allows the instantaneous invention of any form of composition, and which is the key to any kind of structural irregularity.

The following short example illustrates a typical situation, and the issues involved.

Example 3: Selecting Distinct Numbers

Given is a sequence of (not necessarily different) numbers $r_0, r_1, r_2, \cdots$. Select the first n distinct numbers and assign them sequentially to an array variable a with n elements, skipping any number r_i that had already occurred. (The sequence r may, for instance, be obtained from a pseudo-random number generator, and we can rest assured that the sequence r contains at least n different numbers.)

An obvious formulation of a program performing this task is the following:

```
for i := 1 to n do                                   (16)
begin L: get(r);
  for j := 1 to i−1 do
    if a[j] = r then goto L;
  a[i] := r
end
```

It cannot be denied that this "obvious" solution has been suggested by the tradition of expressing a repeated action by a **for** statement (or a DO loop). The task of computing a value for a is decomposed into n identical steps of computing a single number $a[i]$ for $i = 1 \cdots n$. Another influence leading to this formulation is the tacit assumption that the probability of two elements of the sequence being equal is reasonably small. Hence, the case of a candidate r being equal to some $a[j]$ is considered as the exception: it leads to a break in the orderly course of operations and is expressed by a jump. The elimination of this break is the subject of our further deliberations.

Of course, the **goto** statement may be easily—almost mechanically—replaced in a transcription process leading to the following **goto**-less version.

```
for i := 1 to n do                                   (17)
begin
  repeat get(r); ok := true;
      j := 1;
    while (j<i) ∧ ok do
      begin ok := a[j]≠ r; j := j+1
      end
  until ok;
  a[i] := r
end
```

The transcription consists of the replacement of the **for** statement with a fixed termination condition depending on the

running index j by a more flexible **while** statement allowing for more complicated, composite termination (or rather continuation) conditions. But this solution appears quite unattractive. It is admittedly less transparent than the program using a jump, in spite of the fact that the most frequently heard objection to the use of jumps is that they obscure the program. The other objection is that the **goto**-less version (17) requires more comparisons and tests, and hence is less efficient.

The crux of the matter is that well-structured programs should not be obtained merely through the formalistic process of eliminating **goto** statements from programs that were conceived with that facility in mind, but that they must emerge from a proper design process. Two alternative solutions are presented here as illustrations.

In the first case, we abandon the notion that the program must necessarily be based on the statement

```
for i := 1 to n do                                    (18)
a[i] := the next suitable number
```

and consider the basic iteration step to consist of the generation of the next element of the sequence r, followed by the test for its acceptability.

```
i := 1;                                               (19)
while i ⩽ n do
begin generate next r;
   assign it to a[i];
   check whether all a[j] are different
         from a[i];
   if so, proceed by incrementing i
end
```

This form makes it obvious that we are in trouble, if the sequence r should be such that i cannot be incremented any longer. Written in terms of our programming language, (19) becomes

```
i := 1;                                               (20)
while i ⩽ n do
begin get(r);
   a[i] := r; j := 1;
   while a[j] ≠ r do j := j+1;
   if i = j then i := i+1
end
```

The second approach to this problem retains the basic concept of the solution as shown in (18). From there, its development is characterized by the following two snapshots:

```
for i := 1 to n do                                    (21)
   repeat generate the next r;
      check its acceptability
   until acceptable
```

```
for i := 1 to n do                                    (22)
   repeat get(r);
      a[i] := r; j := 1;
      while a[j] ≠ r do j := j+1;
   until i = j
```

In contrast to (20), this solution consists of *three* nested repetitions instead of only two, and therefore seems inferior at first sight. In fact, however, solution (22) turns out to be even more economical. The reason is that in (20) the test for continuation $i \leqslant n$ is actually unnecessary whenever $i \neq j$, since $i \neq j$ in this case implies $i < j$, and because i has not been altered since the last evaluation of $i \leqslant n$. Of course, program (22) is considerably more efficient than the original form with a jump (16).

This terminates our consideration of Example 3.

The question remains open, of course, whether jumps can *always* be avoided without disadvantage. I shall not venture to answer this question, particularly because the term "disadvantage" is sufficiently vague to admit many interpretations. But there is evidence of the existence of some characteristic and reasonably frequent situations which are expressed only with difficulty in terms of the language construct introduced above. A particular case is the *loop with exit(s) in the middle*. Lately it has led language designers to introduce specific constructs mirroring this case [12]. It turns out, however, that it is most difficult to find a satisfactory and linguistically suggestive formulation, and that sometimes solutions are invented that seem to merely replace the symbol **goto** by another word, such as **exit** or **jump**. For example, the construct

```
loop S1;                                              (23)
   exit if P;
      S2
end
```

with the parametric statements $S1$, $S2$, and the termination condition P might be adopted to express the program

```
L1: begin S1;                    (24)
      if P then goto L2;
      S2; goto L1
L2: end
```

in a more concise and **goto**-free form.

Expressing (24) in terms of the basic repetitive statement forms does, indeed, often lead to undesirable complications, such as unnecessary reevaluation of conditions, or duplication of parts of the program, as is shown by the two proposals (25) and (26).

```
repeat S1;                       (25)
  if ¬P then S2
until P
```

```
S1;                              (26)
while ¬P do
  begin S2; S1 end
```

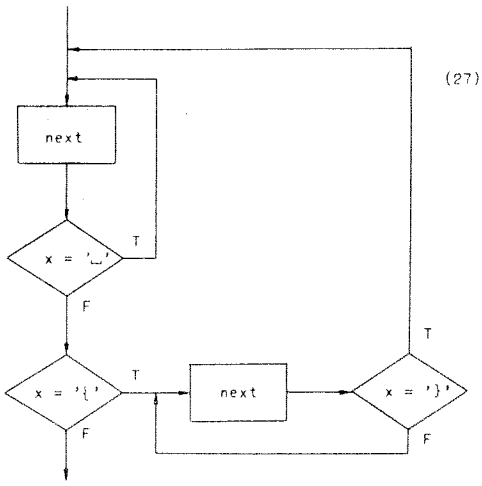
LOOP STRUCTURES

The following, and last two, examples of problems are added to show that often the need for an *exit in the middle* construct is based on a preconceived notion rather than on a real necessity, and that sometimes an even better solution is found when sticking to the fundamental constructs.

Example 4: A Scanner

The task is to construct a piece of program which, each time it is activated, scans an input sequence of characters, delivering as result the next character, but skipping over blanks and over so-called comments. A comment is defined as any sequence of characters starting with a left bracket and ending with a right bracket.

This scanner could typically occur as part of a compiler. A common solution is indicated by the following flowchart (*next* denotes the operation of reading the next character and assigning it to the result variable x).



This program clearly exhibits the loop structure with exit in the middle, satisfying the one-entry-one-exit prerequisite. Instead of proposing a suggestive form for this construct in sequential language, however, let me tackle the posed problem in a different manner. Recognizing the main purpose of the program as being the reading of the next character, with the additional request for skipping over blanks and comments, I propose a first version as follows:

```
next;                            (28)
while x in {'␣', '{'} do
  "skip blanks and comments"
```

The correctness of this program is easily established, assuming that the statement in quotes performs what it says, and nothing more. The definition of the **while** statement guarantees that the resulting value of x is neither a blank nor a comment, no matter in what way blanks and comments are skipped.

The refinement of the statement in quotes is guided by the fact that upon its initiation x is either a blank or an opening bracket.

```
if x = '␣' then next else        (29)
  "skip comment"
```

where the last statement is expressed, with obvious reasoning, as

```
begin repeat next until x = '}';  (30)
  next
end
```

Only knowledge about the expected *fre-*

quencies of occurrence of individual characters can be a reason to choose another form of this program on the grounds of efficiency. For the sake of argument, let us assume that short sequences of blanks are particularly frequent and that, on the other hand, immediately adjacent comments are extremely rare. This leads us to an equally correct alternative form of (29), namely

"skip consecutive blanks, if any"; (31)
"skip comment, if any"

The first of the two statements is readily expressed as

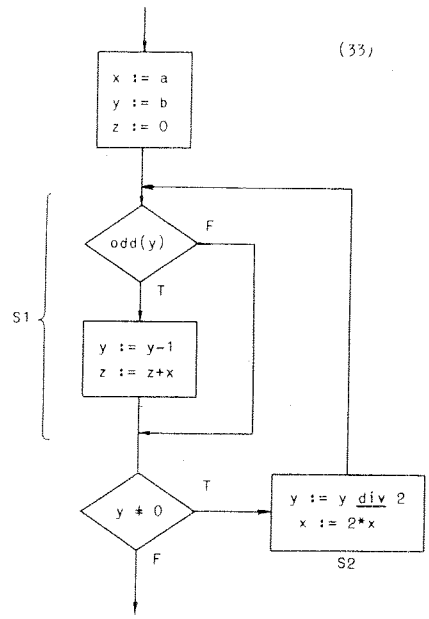
while $x = '\sqcup'$ **do** *next* (32)

whereas the second is already elaborated in (30).

Example 5: Integer Multiplication

Assume that we are to design a program to multiply two non-negative integers a and b with the use of addition, doubling, and halving only. Let the result be represented by a variable z. A well-known and efficient method is shown by the following flowchart:

(33)



This program, once again, clearly exhibits the loop structure with exit in the middle, and therefore cannot be expressed as a single **while** statement. It is usually squeezed into the simple loop form by displacing the loop termination test, positioning it in front of statement $S1$. The program then obtains the well-known form

$x := a; y := b; z := 0;$ (34)
while $y \neq 0$ **do**
  begin if *odd*(y) **then**
    begin $y := y-1; z := z+x$
    end;
    $y := y$ **div** $2; x := 2*x$
  end

This clearly does not change the effect of the program, because if $y = 0$ at entry to $S1$, then $S1$ has no effect and, in particular, leaves y unchanged; and if $y \neq 0$, then the only additional effect incurred by the modified version is on the auxiliary variables x and y in the case of $y = 1$. But this additional effect is quite undesirable, not so much because of the additional, superfluous, and useless computation, but because this operation may be harmful by causing overflow of the arithmetic unit. Should we therefore resort to the exit-in-the-middle version?

A different solution was shown to me by E. W. Dijkstra. He proposed to tackle the problem at its roots, instead of trying to remedy a preconceived proposal. The most obvious multiplication algorithm under the stated constraints is the following:

$x := a; y := b; z := 0;$ (35)
while $y \neq 0$ **do**
  begin $\{y > 0$ and $x*y+z = a*b\}$
    $y := y-1; z := z+x$
  end

Before we start out trying to improve this version, we observe that at the outset of each repetition two conditions are satisfied.

1. $y > 0$ follows from the fact that y is a non-negative integer and not equal to zero.
2. $x*y+z = a*b$ is invariant under the two repeated assignments. (To verify this claim, substitute $y-1$ for y and $z+x$ for z; this yields $x*(y-1)+(z+x) = x*y+z = a*b$, i.e., the original equation.) At entry the equation is satisfied, since $z = 0, x = a, y = b$.

Note that the invariant equation combined with the negation of the continuation condition yields $(y = 0)$ and $(x*y+z = a*b)$, i.e., the desired result $z = a*b$.

If we now insert any statement at the place of the invariant which leaves the product $x*y$ unchanged, the result of the program will evidently remain the same. Such a statement is, e.g., the pair of assignments

$$y := y \textbf{ div } 2; x := 2*x \qquad (36)$$

Under the condition that y is even. But if a relation is invariant over a statement, it remains so regardless of how often the statement is executed. This suggests the following, quite evidently correct, efficient, and elegant solution. It contains no exit-in-the-middle loop.

```
x := a; y := b; z := 0;                              (37)
while y ≠ 0 do
begin {y > 0 and x*y+z = a*b}
  while even(y) do
    begin y := y div 2; x := 2*x
    end;
  y := y−1; z := z+x
end
```

So much for examples, whose purpose was to sketch and elucidate the basic ideas behind the methods of structured programming and stepwise refinement.

CONCLUSIONS

Skeptics will, of course, doubt that these methods represent any progress over the techniques of the old days—in fact, that they are *methods* at all. I can merely say that in my own experience, the new approach has improved my attitudes and abilities towards programming very considerably, and the experiences of others confirm this impression [10, 11]. A systematic, orderly, and transparent approach is mandatory in any sizable project nowadays, not only to make it work properly, but also to keep the programming cost within reasonable bounds. It is the very fact that computation has become very cheap in contrast with salaries of programmers, that squeezing the machines to yield their utmost in speed has become much less important than reliability, correctness, and organizational clarity. It is not only more urgent, but also much more costly to correct an efficient, but erroneous program, than to speed up a relatively slow, but correct program. In the past, the debugging phase has taken a ridiculously large percentage of the development cost in most large projects. The aim now is to eliminate the necessity of debugging by creating bug-free products in the first place. Doesn't this bring to mind the medical slogan "prevention is better than healing"!

The criticism has been voiced that the method of structured programming is in essence nothing more than programming by painstakingly avoiding the use of jumps (**goto** statements). One may, indeed, come to this conclusion by looking at the entire issue in the reverse direction. But in fact, the method of stepwise decomposition and refinement of the programming task automatically leads to **goto**-free programs; the absence of jumps is not the initial aim, but the final outcome of the exercise. The claim that structured programming was invented by proving that all programs can be formulated without **goto** statements is therefore based on a fundamental misunderstanding.

The question of whether jumps enter the picture or not is basically a matter of the level of decomposition or refinement to which the programming process is carried. Ultimately—that is in machine code—there can be no doubt about the presence of jump instructions. The moral of the story is that jumps must not be used in the initial conception of a general algorithmic strategy, and in fact should be delayed as long as possible. With today's state of technology, the introduction of jump instructions can be left to compilers of languages that offer adequate, judiciously chosen, disciplined structuring facilities.

One of the essential facilities for this purpose, besides conditional and repetitive statements, is the *recursive procedure*. In many cases it emerges as the natural formulation of a solution, such as, for instance, in most cases of backtracking algorithms. Hardly anywhere else can a natural, concise, and often self-explanatory solution be made

more obscure and mystifying than by replacing its recursive formulation by one in terms of repetition and—well—jumps. This process should definitely be left to a compiler, as it concerns what is called *coding* rather than programming. (code = system of symbols used in ciphers, secret messages, etc. [Webster].) Modern programming systems, however, offer efficient implementations of recursion, and thereby make "programming around recursion" a largely unnecessary exercise.

Whereas a teacher should not and must not pay attention to "percent issues" as to efficiency while explaining and exemplifying methods of composing well-structured programs, a professional programmer may well be forced to do so. He may sometimes find a dogma of sticking exclusively to a restricted set of program structuring schemas too much of a straight-jacket, and the temptation to break out too powerful. This will be the case as long as compilers are insufficiently sophisticated to take full advantage of disciplined structuring. Naturally, there will always be situations where a compiler is either denied the full information needed for successful code optimization, or where it would be unable to infer the necessary conditions. It is therefore entirely possible that in the future a more interactive mode of operation between compiler and programmer will emerge, at least for the very sophisticated professional. The purpose of this interaction would not, however, be the development of an algorithm or the debugging of a program, but rather its *improvement under invariance of correctness.*

The foregoing discussion also implies an answer to the question of whether structured programming in an unstructured language (such as FORTRAN) is possible. It is not. What is possible, however, is structured programming in a "higher level" language and subsequent hand-translation into the unstructured language. The corollary is that whereas this approach may be practicable with the almost superhuman discipline of a compiler, it is highly unsuited for *teaching* programming. Recognizing that there may be valid economic reasons for learning *coding* in, say, FORTRAN, the use of an unstructured language to teach *programming*—as the art of systematically developing algorithms—can no longer be defended in the context of computer science education. The lack of an adequate modern tool on the available computing facility is the only remaining excuse.

The last remark concerns an aspect of "structured programming" that has not been illuminated by the foregoing examples: structuring considerations of program and data are often closely related. Hence, it is only natural to subject also the specification of data to a process of stepwise refinement. Moreover, this process is naturally carried out simultaneously with the refinement of the program. A language must, therefore, not only offer program structuring facilities, but an adequate set of systematic data structuring facilities as well. An example of this direction of language development is the programming language PASCAL [12, 13]. The importance of this aspect of programming is particularly evident, as we recognize the data as the ultimate object of our interest: they represent the arguments and results of all computing processes. Only structure enables the programmer to recognize meaning in the computed information.

ACKNOWLEDGMENT

The author is grateful to P. J. Denning for kindly posing the problem treated in Example 3.

REFERENCES

1. DIJKSTRA, E. W. "Some mediations on advanced programming," *Proc. IFIP Congress 1962*, North Holland, Publ. Co., Amsterdam, The Netherlands, 535–538.
2. DIJKSTRA, E. W. "The humble programmer," ACM Turing Lecture 1972, *Comm. ACM* **15**, 10 (Oct. 1972), 859–866.
3. DIJKSTRA, E. W. "Notes on structured programming," in *Structured Programming* by Dahl, O. J.; Dijkstra, E. W.; and Hoare, C. A. R., Academic Press, New York, 1972.
4. NAUR, P., AND RANDELL, B. "Software engineering," Report on a conference sponsored by NATO Science Committee, Garmisch, West Germany, October 1968.
5. WIRTH, N. "Program development by step-

wise refinement," *Comm. ACM* **14,** 4 (April 1971), 221–227.

6. WIRTH, N. "Systematic programming," Prentice-Hall, Inc., Englewood Cliffs, N. J., 1973.
7. NAUR, P. "Proof of algorithms by general snapshots," *BIT* **6,** 4 (1966), 310–316.
8. HOARE, C. A. R. "An axiomatic basis for computer programming," *Comm. ACM* **12,** 10 (Oct. 1969), 576–581.
9. WULF, W. A. "Programming without the **goto**," *Proc. IFIP Congress 1971,* Vol. I, North Holland Publ. Co., Amsterdam, The Netherlands, 1972, 408–413.
10. BAKER, F. T. "Chief programmer team management of production programming," *IBM Systems J.* **11,** 1 (1972), 56–73.
11. AMMANN, U. "The method of structured programming applied to the development of a compiler," *Proc. Internatl. Computing Symposium 1973,* North Holland Publ. Co., Amsterdam, The Netherlands, 1974, 93–100.
12. WIRTH, N. "The programming language Pascal," *Acta Informatica* **1,** 1 (1971), 35–63. 35–63.
13. JENSEN, K., AND WIRTH, N. "PASCAL—user manual and report," *Lecture Notes in Computer Science,* Vol. 18, Springer-Verlag, Berlin, Heidelberg, New York, 1974.