The Basic Principles of Well-structured Code

By Arthur Sale*

This tutorial paper sets out to explain the basic principles underlying the concepts of well structured code, and the reasoning behind these principles. The alleged advantages of well-structured coding include increased programmer productivity, less debugging, etc., and it is therefore of potential interest to a wide range of computer professionals and users. The general topic of structured programming is also discussed to put well-structured code into perspective in the overall concept.

1. BACKGROUND

The term *structured programming* often evokes extreme reactions in computer programmers, either of active dislike or fanatical acceptance. Why this is so is an interesting question in the history of computing, and perhaps also important for our understanding. The roots of structured programming lie far back in the development of the so-called software crisis. We now recognise that the problems of large software projects related not so much to crisis as to impasse (unless we had mistakenly accepted a contract for an overlarge project), and that some large software projects were unwritable however much money was poured into them. The ways to overcome this problem seemed to lie in dramatic improvement in programmer ability, or in greatly improved organizational techniques, but no-one seemed to have the answer.

The first major push towards what we now call structured programming came from Professor Edsger W. Dijkstra of the Technological University of Eindhoven who suggested at the 1965 IFIP Congress [Dijkstra, 1965] that the **goto** statement should be eliminated from high-level programming languages. This comment, repeated and expanded later in a letter to the Communications of the ACM [Dijkstra, 1968a], triggered off a number of interested researchers, but had relatively little effect upon the industry itself. The idea of avoiding the use of **gotos** had of course been around for quite some time; this early history is well-catalogued by Knuth [1974]. Perhaps some of the failure to listen can be attributed to the provocative tone of the letter, perhaps some to the mislcading and negative title placed above the letter; but much more importantly, it seems the idea was generally rejected because it did not fit the accepted model of what programming activity ought to be like. In Kuhn's terms [Kuhn, 1962], the existing *paradigm* of programming was a view of creating intricate programs of a certain personal beauty of complexity bound up with a skill in diagnosis of faults. As a result, only those persons who had personally experienced some of the problems and perhaps were less than entranced with the skills of the allegedly best programmers were able to grasp that here perhaps was the start of a new paradigm which might be employed to motivate programmers: *to employ structure of simple kinds in a regular manner to construct reliable programs.*

Dijkstra meanwhile set himself to work intensively on a self-set project:

> "This working document reports on experience and insights gained in programming experiments performed by the author in the last year. The leading question was if it was conceivable to increase our programming ability by an order of magnitude and what techniques (mental, organizational or mechanical) could be applied in the process of program composition to produce this increase . . ."
>
> [Dijkstra, 1970]

This project led to a large number of seminal papers and quotable phrases as his ideas matured and developed [Dijkstra 1965, 1968a, 1968b, 1969, 1970, 1971, 1972 and others] [Dahl, Dijkstra and Hoare 1972]. Dijkstra undoubtedly provided the impetus and thrust which led to the eventual recognition that there must be something useful in structured programming.

It would be invidious here to single out all the contributions made in the field (not least for fear of omitting some important development), but it should be mentioned that two other workers who contributed considerably to the general climate of tolerance of these ideas were Professor C.A.R. Hoare and Professor N. Wirth. Their emphasis on what can now be called software engineering (without too much risk of understanding) altered the programmer's view of his activity. No longer is it appropriate to view a programmer as a lone eccentric pursuing an esoteric occupation (though this misconception persists amongst the young), but rather as a designer attempting to employ his skills in creating useful human constructs in association with others . . . Further bibliographies can be found in Knuth [1974] and Lecarme [1974].

A year or two ago, to the surprise of those of us who had followed the earlier developments and were thinking about the consequences, the situation suddenly changed: it became fashionable to talk about structured programming; it was the in-thing to do. And naturally (as with all fashions) it rapidly got distorted and gathered accretions of bandwagon topics, for example IBM's superprogrammer project, and Dijkstra's own further ideas on top-down designing. The situation exhibits all the characteristics of a small *scientific revolution* [Kuhn, 1962] in progress: the accepted paradigm is changing and we witness all the characteristics of crisis science: diehards clinging to old beliefs; confusion and redefinition of terms; the obsolescence of many text-books; and the many proponents taking dogmatic positions.

As an almost inevitable result of the revolution, numerous people have elevated the discussion of structured programming to an almost theological level. Authors are often violently polarized one way or another and take up positions from which it is difficult to budge them. Dijkstra

*Department of Information Science, University of Tasmania. *Manuscript received 5th May, 1975.*

has commented on a communication to Knuth [Knuth, 1974] that not only has he received a considerable amount of crank mail, but that the position in the controversy ascribed to him is somewhat more extreme than he cares for.... In the process catch-phrases achieved notoriety in attempts to gain attention. For example the very useful word structure has almost been devalued to meaninglessness today, and the catch-phrase goto-less programming has focussed much attention on one negative aspect of what the revolution is all about. Often this has ensured a negative response from active programmers, for who wants to be banned from using a tool because they're felt to be too irresponsible? It has accordingly become wise for you to get any self-confessed structured programming proponent or opponent to define what he means by the terms before you listen further.

2. THE OBJECTIVES

What then is structured programming as I see it (for inevitably the discussion will be coloured by personal views)? Structured programming is about *structure* in *programming*: its key precept is that a programmer or system designer needs to be *fully aware* of the implications of the structures he employs in creating programs and their nature, and ought to be able to both *control* and *exploit* these structures in an efficient way. Viewed in this way we can perceive several facets of this central theme, all of which merit extended treatment (and of course further research). For example there is the problem of perceiving structure in the problem itself, and of how this structure might be reflected into the modularity of the program; or the problem of natural or desirable data types and data structures and their associated natural manipulations; or the problem of coding actions together to form a program. In this paper, concentration will be focussed on this last topic: *of how program code ought to be put together and of the advantages of controlling the code structures employed.*

I could now go into some detail to point out the various advantages of and reasons for employing these techniques, but this would probably not convince or satisfy those who have not met structured code before. I will however point out that the alleged benefits of structured code are:

* modularity of the code is enhanced so that local changes do not propagate effects all over the program,
* it becomes easy to satisfy oneself of the logical correctness of a program with an amount of effort which grows only linearly with the size of the program (and does not result in an explosive increase of effort in large programs), and
* maintenance and modification become simpler as the structures employed are simple and regular.

In emphasizing the code structure aspect, I do not wish to play down other structuring areas, but simply to concentrate on one isolatable component of a good programmer's toolkit. I have not space here to explore the possibilities of data structuring and data types, nor to write much about the woeful deficiency of most languages in this respect. Suffice it to say that a well-structured language will let you specify data and operations thereon in terms of the natural properties of the objects concerned, regardless of the fine details of implementation, and ought to protect you from violating the rules implied by the objection properties. Floating-point arithmetic is a reasonably good example of what is wanted; you are given natural operations (+, −, x and ÷), are fairly isolated from the actual bit-patterns and algorithms used, and in some computers are notified of overflow and underflow violations. Even here there is room for improvement in practice, as witnessed by Knuth's comments [1969] on arithmetic axioms and Wirth's criticism [1972b] of CDC 6600 arithmetic. But to take a very simple example where typical languages break down, consider the days of the week: Monday, Tuesday, Wednesday, ..., Sunday. If we have a variable which is to hold a value of this type, then clearly an integer is not the best possible representation. After all, the days of the week are circularly ordered (not linearly) and violations of the implicit bounds will not be reported if we try to find successors to each day. And anyway why should a programmer have to decide whether Monday is represented by a 0 or a 1? Or indeed whether the week starts with Monday or Sunday? The whole mess arises because common languages only have a few very simple data types and structures, and very minimal checking. Corollary: most computing langauges are dreadful: we are only now beginning to realize quite how bad and what ought to be done about it [Hoare 1972].

To close this introductory section, I will quote a number of other definitions of structured programming. It should be clear from all these just how much confusion exists in detail and yet how all are groping towards similar things...

> "The systematic use of abstraction to control a mass of detail, and also a means of documentation which aids program design."
>
> [C.A.R. Hoare in Knuth 1974]
>
> "The major ideas that we group under the heading of structured programming comprises: complete or partial banishment of the **goto** statement, by way of logical constructs with nested structure; a novel approach to modularity; construction of programs by stepwise refinement; top-down programming; analytic verification or proof of correctness of algorithms; and in the use in program construction of a strict but freely accepted discipline."
>
> [Lecarme 1974]
>
> "A major function of the structuring of the program is to keep a correctness proof feasible."
>
> [Dijkstra in Dahl, Dijkstra and Hoare 1972]
>
> "The purpose of structured programming is to control complexity through theory and discipline."
>
> [Mills 1973]

For further comment on a rational level, see the letter in ACM Forum by Gries [1974].

3. CODE PRIMITIVES

Translated into action, the key precepts of structured programming imply that programs ought to be constructed from code structures that we understand, and whose inter-relations are themselves easy to understand. Accordingly we are looking for code structures which are simple, and in some sense most basic: a set of code primitives. Note that we are not seeking for an absolutely fewest set of primitives from which to construct all programs (otherwise we might land up with a Turing machine), but rather to find a consistent set of structures that achieves a balance between simplicity of structure and interaction and naturalness of use. The necessary compromises will naturally allow slightly different points of view to be legitimately held and argued...

Let me now postulate that good programs can be written using only the following set as basic primitive code structures:

* any indivisible action which has an effect defined by associated axioms, one entry point (inway) and one exit point (outway).
* the composition rule.
* two enumerative combination rules:
  (1) sequential,
  (2) selective.
* two repetivie uses of code:
  (1) the iterative loop, and
  (2) the self-recursive procedure.
* the complier/interpreter/table-driven program structure, which in effect defines a new abstract machine with new axioms for a higher level of code.

With the exception of the last construct, all of these have but one inway and one outway, and consequently only one (albeit complex) action. The last construct is different, in that it defines a *new* set of axioms as a basis for a *new* body of code, and the flow of control in the interpreter (or compiler or whatever) has little relation and no relevance to the flow of control in the new body of code.

In discussing these primitives, it is appropriate to begin with the *basic indivisible action*, which we take as being defined by the language we are using, or better, by the natural properties of the objects being manipulated. The effects of such a basic action are to cause some change in the data environment or the external environment, the nature of this action being specified by some properties or axioms. If a boundary assumption concerning the action is violated, we have a right to expect that we be notified immediately of the fault, and if the violation is not detected until run-time, that either the execution is terminated or some other remedial action can be brought into play. Some simple examples of basic indivisible actions in a hopefully self-evident language are:

```
RESULT ← A x (BV + 1)
PRINT RESULT, "CONVERGED"
```

At this point it might be appropriate to mention that some people seem to have a mental block for words like proof and axiom, probably due to some unfortunate experiences during early mathematical learning. Although I shall continue to use these words (they have meaning for me and have the merit of being short), you are quite at liberty to mentally substitute 'satisfy oneself of the logical correctness of' for 'prove', and 'principles of operation' for 'axiom' wherever these words occur. You will miss no nuances of meaning.

The composition rule, which I introduce next, simply states that any given code construct with one and only one inway and one and only one outway, may be treated as a basic action for the purposes of creating larger structures, however complex the given construct may be internally. You are undoubtedly familiar with this rule as it crops up in most common programming languages in two guises: that of nesting (which directly encloses a construct within another) and that of procedure calls (which effectively achieve the same control effect). Macros are yet another manifestation of the same rule, the differences between these implementations of the composition rule lie primarily in the different treatment of the data- and external-environments. A nested construct (for example a loop within a loop) has access to all the data-objects available to the outer construct, whereas in the case of a procedure (or function or subroutine or subprogram) access to the data-objects of the invoking code may be restricted or directed. A macro of course differs from a procedure primarily in its run-time attributes of time and space, at least in normal usage.

The importance of the composition rule is that all the code constructs I shall consider will satisfy the required one inway and one outway properties, and consequently the composition rule can be used to make structures of considerable complexity and widely different substructure. This rule probably requires little more explanation as it is so widely known, and the following few examples should suffice:

Nesting:

```
LOOP FOR J ← 1 TO N
    LOOP FOR K ← 1 TO N
        ARRAY [J,K] ← J x N + K
    END LOOP
END LOOP
```

Procedure:

```
PROCEDURE LOGBASETWO(X, RESULT)
    ...
    RESULT ← LOG(X) / LOG(2.0)
END PROCEDURE
PROCEDURE MAINPROGRAM
    ...
    LOGBASETWO(XC-CELLSIZE, LDELTA)
    ...
END PROCEDURE
```

Macro:

```
MACRO ACCUMULATE(A,B)
    ...
    B ← B + A
END MACRO
PROCEDURE MAINPROGRAM
    ...
    ACCUMULATE(NETPAY, SALARYBILL)
    ...
END PROCEDURE
```

4. ENUMERATIVE CONSTRUCTS

The enumerative constructs are rules for assembling basic actions into larger assemblies of a regular structure. They are characterized by the necessity of enumerating and examining each of the effects of the composing actions one-by-one before the total action of the construct can be comprehended. Consequently the complexity of an enumerative construct depends upon the number of actions making it up. An example will make this clearer.

The most familiar enumerative construct is that of *sequential combination:* of actions executed sequentially one after another. Practically all computing languages permit this form of combination, and undoubtedly you are familiar with it. Clearly to understand the effect of a piece of sequential code you need to understand what the first action does to the environment, then what the second does, and so-on until the last action is considered and the total action of the piece of code is understood.

The topological dual of serial combination of actions is that of combination in parallel, and were I considering actions in a more general sense, it would be necessary to consider the claim of fully parallel actions (in time) as a

primitive. However most programming takes place in a mode where there is a single flow of control (uniprogramming) and only one action at a time is activated. If in this case we have actions where in fact we do not care about the order of execution of the actions, we cannot execute them in parallel, but must combine them in some more or less arbitrary serial order. This state of affairs is now so commonplace that programmers are unused to thinking of actions in parallel, or even of considering which serially combined actions do not in fact interact.

There is however one situation where topologically parallel control paths find a place in a uniprogramming environment: where one and only one of the actions is selected for execution according to some predetermined decision. This is the *selection rule*, and is poorly treated in many common languages. Perhaps the commonest forms of the selection rule are to be found in the **if-then** and **if-then-else** constructions made popular by Algol 60. Consider the **if-then-else** construct represented as follows (in an invented language):

```
IF logical-expression
    action-1
ELSE
    action-2
END IF
```

Depending on the value of the logical expression, one and only one of action-1 or action-2 is executed; action-1 being selected if the logical expression has the value **true**. As we know logical expressions have only two possible values (**true** or **false**) and there are therefore two ways of looking at this construction. Both are computationally equivalent, but they differ in how we view extension of the principle. The first view sees the logical expression as producing one of two values which selects one of two actions; the second view sees the logical expression as producing either a value (**true**) which selects the first action, or any other value (which we know here must be false) as the second default action.

The second view corresponds most exactly to the subconscious view held by most programmers, and is perhaps marginally richer if we look at extensions of the selection rule. The first we might look at is the common restriction to an apparent selection of one action: the **if-then**. If no action is necessary for the default case, then many languages permit a contraction of the construction to the equivalent of:

```
IF logical-expression
    action-1
END IF
```

Here it is understood that the default action is null: do nothing but carry on. That this view is commonly held is shown by the fact that the apparently symmetrical case (with the other viewpoint) of an **if-else** construct is almost never met: programmers prefer to reverse the direction of their logical condition.

Much more interesting though is the extension to richer selections. There is no good reason why a selection should stop at two-way; after all serial combination of arbitrary numbers of actions is commonplace. It has been sometimes argued that since n-way selections can be built up of two-way selections (**if-then-else**), that it is therefore somehow *immoral* to use n-way constructions. This argument is very weak for the same is true using one-way selections (**if-then**) and no-one seems to argue that we be restricted to them. . . In any case the argument is of the same sort as the insistence in some of the New Maths that addition is a **binary** operator despite the common counter-evidence of column addition sums in their own textbooks. In view of the fact that not many languages have fully faced up to the selection question for logical expressions, we will not be surprised to find that they are woefully deficient in n-way selection. We need however the facility of being able to write something like this:

```
SELECT expression
CASE constant-1
    action-1
CASE constant-2
    action-2
CASE constant-3
    action-3
ELSE
    action-4
END SELECT
```

In this form (which is even now not the most general in specifying the selection) the expression selects one of the actions action-1, action-2 or action-3 if it evaluates to match one of the appropriate case constants, or if it does not match, it selects the default action following the ELSE. Like simpler constructs the ELSE may be omitted, and any number of CASEs is permitted. It is possible then to view the **if-then-else** as a convenient shorthand for:

```
SELECT logical-expression
CASE true
    action-1
ELSE
    action-2
END IF
```

Despite the relative unfamiliarity of the selection construct, it should be clear that it too requires enumerating the effects of all alternative actions before the total effect of the construct can be understood. The effort involved, as before, grows with the number of alternative actions. Both sequential and selective combination rules can be shown well in flow-charts as the necessary flow-of-control properties are well modelled, and Fig. 1 illustrates the two structures.

5. LOOPS AND RECURSION

Loops and *recursion* play an important part in computing, for they permit code fragments to be repeatedly used. Indeed, beginning programmers often seem to go under the impression that the smallest interesting procedure worth writing must have at least one loop. Personally, I can think of many useful and complex loop-free procedures that I have written and no doubt you can too; however this feeling has a germ of truth for it is at least true that to understand a loop you require quite different techniques from those used with the combining rules. In fact the way that we understand the operations performed by a loop goes by the technical name of *mathematical induction*, which I shall discuss briefly later.

The most primitive loop (with the fewest

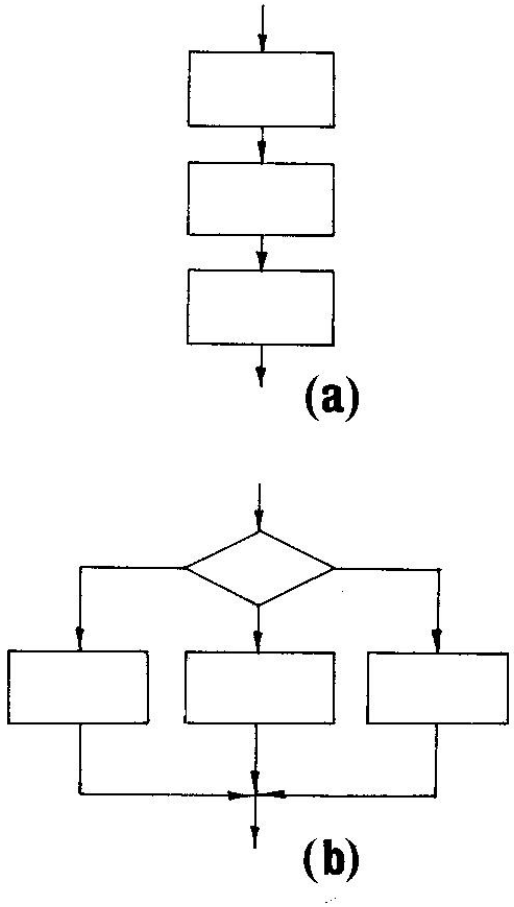

Figure 1 The enumerative combination rules: (a) sequential, and (b) selective.

complications such as flow junctions) is illustrated in Fig 2: the test-at-loop-start version, often characterised in the literature as the **do-while** loop. Following the notation I have been using, such loops could be written in one of the following two forms:

```
LOOP IF logical-expression      LOOP UNLESS logical-expression
    action                          action
END LOOP                        END LOOP
```

The only difference between these two forms is of course in the inversion of the logical expression of the looping condition. Since I am generally more interested in the conditions for escaping from a loop than the conditions for continuing in it, I prefer the second form for clarity. Whether this is true of other programmers or not is uncertain; the commonly suggested **do-while** form of the construct suggests otherwise. However since the evaluation of the logical expression is of no consequence in understanding the action of the loop traversal (in the absence of side-effects it makes no changes in the data-environment), and the expression is therefore primarily of use to the programmer in

(1) determining an assertion about the data-environment that can be made at that point, and
(2) determining the escape condition,

there is room for both points of view.

Most languages have a built-in *count-loop* construct such as BASIC's FOR and FORTRAN's DO; indeed many languages have no other explicit loop construct. Obviously count-loops are important, and it is satisfying to find that a typical count-loop can be viewed as simply a convenient macro-form for specifying a type of do-while loop. Thus the following expansion contains the essence of count-loop action.

```
                                J ← M
LOOP FOR J ← M TO N             LOOP UNLESS J > N
    action                          action
END LOOP                            J ← J + 1
                                END LOOP
```

There are often annoying and unnecessary implementation and/or language details which obscure or limit the mapping, but these usually invoive the manner of handling the data-environment, or the manner of evaluating the loop parameters. In some cases the restrictions placed are such as to simply act as a safeguard: restricting a count-loop to counting so that much more complex usage do not masquerade under this guise. For example it would probably be generally agreed by persons who have examined the problems of count-loops in languages that:

(1) the counting variable ought not to be otherwise changed during the execution of the loop, either in value or in reference;
(2) the limits and step size of the count ought to be fixed similarly throughout the loop execution (so that it truly counts);
(3) the counting variable ought to be either fully defined in value after completion of the loop in a normal way, or else totally inaccessible after completion (thereby evoking a compile error if a subsequent attempt is made to access it); and

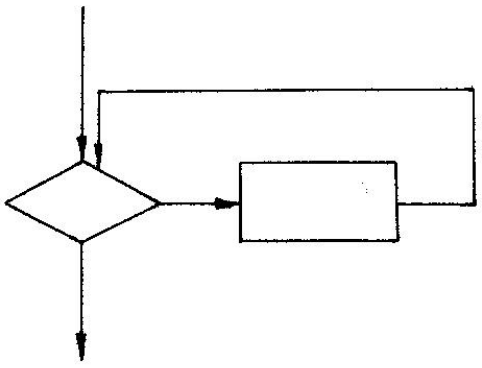

Figure 2 The most primitive loop: do-while

(4) the loop test should be logically at the head of the loop so that it is tested before entry into the loop.

Notice that all these act so as to limit the count loop to its intended purpose and prevent unexpected effects.

Some proponents of structured coding, including Dijkstra and Wirth, have implied that the only loop simple enough to be used in handwritten code is the **do-while** loop (and the count-loop). I believe this to be an extreme and untenable position, as an examination of the code produced by good programmers soon shows. Frequently a situation arises where one of the most natural ways to program is to create a loop which has *two* (or more) conditions for escape from the loop, often at different points in the code for the loop action [Peterson, Kasami and Tokura 1973]. To be sure, such needs can be met by a simple **do-while** loop, but this will normally call for the introduction of auxiliary (logical?) variables whose sole purpose is to hold information until the one termination condition of the **do-while** loop is encountered. Such additional burden on the state description of the loop is only warranted if in fact loops with several points of termination from the loop are in fact too complex to understand. And in fact they are not, at least up to a moderate level of complexity. Accordingly, we need to be able to describe loops with termination conditions at the start of the loop, or at the end of the loop, or somewhere in the middle, or some mixture of all of these. To permit this, I will introduce (a) the ESCAPE statement which causes control flow to terminate the immediately enclosing loop and transfer to the code just after the END LOOP Marker, and (b) as a convenience IF and UNLESS clauses may be attached to any one of the loop marker, the END LOOP marker, or an ESCAPE. A more complex loop might then look something like:

```
LOOP
    ...
    ESCAPE IF A ≠ B
    ...
END LOOP IF J + 1 ⩾ N
```

To understand the action of a loop, it is necessary to establish two basic facts or assertions. Firstly it is necessary to show that *if* some property of the data-environment is true when a loop traversal is about to commence, then it follows that it is true immediately after the loop traversal. Secondly it must be established that this property is true on entry to the loop. This allows us to employ the reasoning technique which goes under the technical name of mathematical induction: the property is true before the looping starts (by assertion 2), and it is therefore true after the first traversal (by assertion 1), and true after the second traversal (by assertion 1 again), etc. It remains only to apply the special conditions of the loop termination condition in the light of the loop assertion to determine the state of the data-environment when the loop is complete and whether it does ever terminate. For obvious reasons the assertion about the data-environment which does not change during the execution of the loop is known as the *loop invariant*. Since loop invariants can be quite complex, most of the difficulty of understanding a loop lies in identifying just what the loop invariant is. An example will be shown later.

So far I have concentrated upon loops as the structure most familiar to readers, and ignored *recursive procedures*. Recursive procedures are simply another way to repeat code controlled by an escape condition and it is regrettable that they have become surrounded by an aura of mystery for the ordinary programmer. Since loops and recursion have much in common, I could cut the discussion of recursion short at this point were it not for the fact that the problems of recursive procedures do not appear to have been discussed anything like as fully as loops and many misconceptions still remain.

In their own way, languages that permit unrestricted use of recursion (such as Algol 60) open the way to a spagetti-bowl style of programming quite as nasty as that composed by unrestricted use of **gotos** (typical of FORTRAN programs). The only thing that generally saves the situation is that free use of recursive procedures is not common, perhaps because they are not required in most programs, or perhaps because they are not understood. As a result the problem of poorly structured recursion rarely arises. When it does, the results can be extremely mystifying as the creation and deletion of local data-environments with recursive procedures can destroy much of the evidence of malfunction. . .

In principle there is nothing objectionable in a procedure (subprogram) that calls itself to evaluate or analyse a sub-program that it has analysed: this is much the same usage as that of a loop, except that the local data-objects are recreated afresh for the new activation of the code (thus partially isolating the new activation from the old). I shall call such procedures *self-recursive*. Self-recursive procedures are certainly well enough structured for us to admit them to our set of primitives: they have to have a termination condition for recursion, a recursion invariant, and require the technique of mathematical induction to understand their effects. In fact self-recursive procedures are as alike to loops (and therefore can be regarded as a slightly different implementation form of the basic principle of reusable code) as procedures are to nested code. It is true for example that any recursive computation can be performed iteratively by a loop, and vice versa (Cooper 1966 and Barron 1968).

Having stated that self-recursive procedures are well-structured since they can be considered in isolation, I should examine the other usages that are commonly lumped under the umbrella of recursion. The first, which is innocuous, involves a reactivation of a piece of code (before its first activation is complete) to analyse a completely different data-environment. A typical example is that of a double integration in numerical work, where the work of evaluating the outer integral involves calling the integration routine to evaluate a quite different problem. Such usages are legitimate and better thought of as reusable or re-entrant code rather than a recursive, even though the same implementation structure may be used. The danger in recursion lies rather in procedures which are at least mutually recursive, or involve more complex calling structure. A common example that springs to mind is a syntax-analyser for a high-level language which permits quite complicated expressions. If there is say an arithmetic-expression analyser routine, and a logical-expression analyser routine, then if logical expressions can be embedded in arithmetic expressions and vice versa, the two routines may call each other to analyse sub-expressions. Such a situation immediately causes the problem of correctness of the routines to be bound up together: it can never suffice to show that each routine is correct on the assumption that the other is, but the pair

must be considered together. A very simple example is presented in the appendix to illustrate this point since it does not seem to be widely known.

Of course this is precisely the situation in which **gotos** force increased effort of understanding and difficulty of proof of correctness. It may therefore be appropriate for those who decry the use of **gotos** but who delight in recursion to re-examine their principles, the situation may not be as black-and-white as was first supposed. If leaping into and out of loops should be banned then so should mutual recursion (for all recursive uses can be formed from self-recursive procedures); if however mutual recursion is sometimes valuable there may even be a case for the occasional **goto**. . .

6. ABSTRACT MACHINE CREATION

The effect of the composition rule and procedures in particular is to extend the available code actions to a more suitable set for the higher level programming task. Consider a situation where, by some mechanism or other, a complete set of new actions has been defined, and the higher level code is written entirely in these actions and not in the lower level actions of which they are made up. Imperceptibly, we have moved from a position of extension of the code structures to that of redefinition of available code structures: the *creation of a new abstract machine.*

While there are sometimes cases where it is difficult to decide whether a programmer is extending a machine or creating a new one, there is more often no doubt that the creation is real. Notice that my usage of the term *machine* embraces the term *language*, with the additional connotation that not only can we write code for the machine (language) but that it can also be executed. The importance of this mode of coding is that by creating a new set of axioms and actions which are more suitable for the description of the problem, the problem itself becomes easier, and perhaps better defined. The trick of designing of course involves a good choice of what new level to construct, and what properties it ought to have. Some programmers may be of the opinion that this sort of programming is reserved for esoteric applications, but this is not wholly correct. To illustrate the different types of application and implementations, I shall briefly discuss macro- and subroutine packages, compilers and interpreters, and table-driven programs.

It is quite common to find instances of extensive sets of subroutines provided for some particular application (for example FORTRAN routines for commercial data processing applications) which require the writing of driver programs which consist entirely of calls to routines. In another case, macros are used to define a new language which is then used to write a program. This approach has been put forward by Waite [1970a, 1970b] as a partial solution to the portability problem. In each case the user of the system uses a language which involves objects more directly useful and familiar to him (though the same may not be true of the notational devices).

A more whole-hearted approach is of course to create a whole new code environment, including actions, syntax, data-objects, etc. This is exemplified in emulators (microcode creation of fake machines as for example IBM 370s), in simulators and interpreters (apparently direct execution of higher-level languages), and in compilers (manipulation of the higher-level language to make it more palatable to the lower level execution device). And we all know of cases of compilers compiling other compilers, which perhaps run on machines which are themselves emulated by a host processor. . . . Clearly the heirachical nature of these levels is very desirable, as each level can be considered in isolation from the others [Dijkstra 1969].

However this type of coding is also of use (and common) in simpler situations: typically of the so-called table-driven programs. Examples that spring to my mind are the syntax tables of many compilers (such as the WATFOR

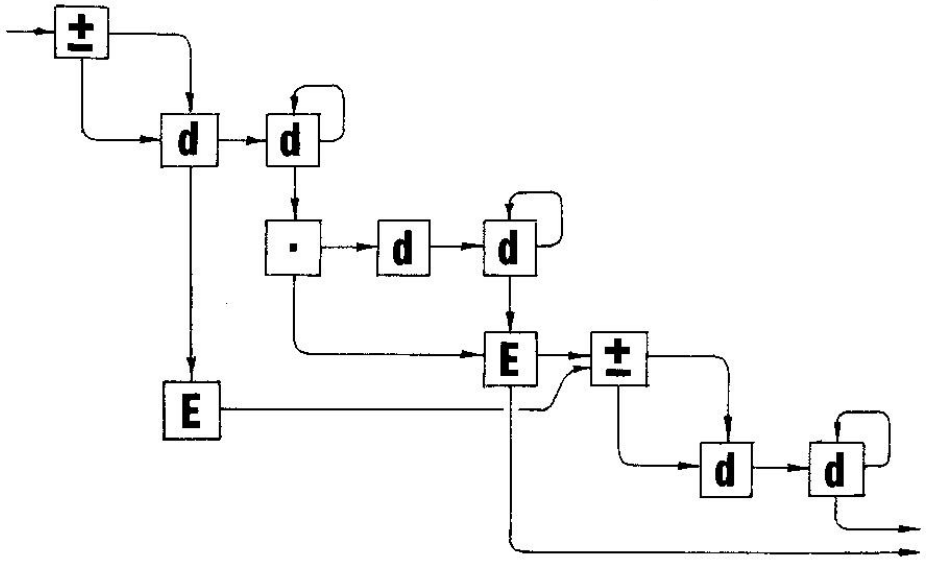

Figure 3 Character syntax for floating-point numbers (see text for explanation of conventions)

compiler) and decision tables. The way of thought involved is sufficiently important to warrant an example.

Consider the problem of recognising floating-point numbers as typically presented to FORTRAN, BASIC, Algol and PL/I systems, either in the source code as constants or in the input data. One way of recognising correct floating-point numbers is to sit down and carefully work through the syntax rules, finally writing a well-structured program (full of **if-then-elses** and **do-while** loops) to explore all the possible cases. This approach was followed in Algorithm 239 published in the Communications of the ACM [McKeeman, 1964]. The result can be inspected and while it generally conforms to the rules of well-structured programs, it is unnecessarily large and difficult to understand, running to some 61 non-comment lines of Algol 60. An alternative approach is to create a fixed data-structure (table) which reflects the syntax structure, and then to write a program that simply 'interprets' the table by making appropriate tests and conditionally changing its state to a new point in the table. Such an approach leads to an executable code body which may be much shorter (in a trial implementation: 27 lines of Algol-like code), and is much different. It of course implements a quite different problem from the first, and its flow-chart affords no certain guide as to the action of the total procedure. The essence of the design is to concentrate its control in the table, which is therefore a higher level description of the desired action. You can understand, too, the advantages of this implementation in adaptation to the differing vagaries of syntax in Algol and FORTRAN, etc.

To illustrate this further, Fig 3 shows graphically a typical table structure which specifies the syntax of a floating-point number in some language. Each node consists of characters to be tested against the current input string character (± means + or −, and *d* means one of the digits 0 to 9, and *E* and . mean themselves), and two pointers to be taken dependent on the success of the match: right across the page if a match occurs (whence move on to the next character in the input string), and downwards if no match occurs (so keep trying). In trying to match a string if you reach the right hand side you have analysed a number, and if you reach a downwards missing link then the string is syntactically not a floating-point number. Of course many other equivalent syntax structures are possible for this problem.

7. TO GOTO OR NOT TO GOTO

One of the tenets of structured programming is then that any desired program action can be and ought to be implemented without excessive efficiency loss using only the code primitives discussed earlier. That all possible programs can be implemented in this set is very easy to prove: one loop and one wide selection suffices to implement all programs though with poor efficiency and quite unacceptable loss of structure related to the problem. In general, the claim of only marginal loss of efficiency (if any) is reasonably true: the structures available fit most programming usages. Remember though that some proponents of structured programming advocate a much more restricted set of primitives than I have discussed: a somewhat greater efficiency loss may be the result. Slower running may however not be the normal penalty for writing structured code since most complex constructions beloved by tricky programmers turn out to be of negligible importance in overall program execution time; the innermost loops of programs which consume most of the time usually turn out to be pretty simple things.

What then about the **goto**, familiar to every FORTRAN, BASIC and assembly language programmer? Has it any role left? at this stage, the answer must be clearly yes, on two counts. The first can be disposed of quite quickly: the **goto** or jump or branch has a very important role to play in machine instruction sets as it is such a powerful and economical implementation technique for changing control flow. Machine designers will be loth to give it up, and they need not. Obviously therefore the **goto** may similarly crop up in intermediate object codes, and in low-level assembly languages (though there is a case for encouraging structured code even in die-hard assembler programmers). *No; the* **goto** *has a place deep in machines: it is misuse by humans that is objected to.*

The second role for the **goto** is much more controversial. Some programming practices turn out to be not very easily accommodated by the primitives proposed earlier. Examples that come to mind are escapes from many levels deep in nested loops (the loop primitive was restricted to one level of escape), and **gotos** on special exception conditions (error discovery, or end-of-file). Should such usages be regarded as poor programming and other constructions be substituted? Or should the usages be tidied up by a better form (an **event** notice has been suggested [Knuth 1974]). Or is the **goto** here the best implementation? No-one really knows as yet, and until the question is resolved, the **goto** remains a convenient mechanism for occasionally constructing a complex code structure or for attaining efficiency improvements in time-critical code.

It is important to keep perspective on these issues: structured programming is about designing and perceiving structure in programs and in relating that structure to the problem and its related sub-problems. It is useless conforming to all the rules of structured code if the resulting program is not clear and well-designed: program structure can hide as well as illuminate problem structure... In short, it is possible to write terrible programs with structured code as it is with **gotos**; only we hope much less likely!

8. CORRECTNESS PROOFS

I can now turn to discussing some of the benefits of structured code. One of the prime benefits claimed is that of being able to prove that program segments are correct; of being able to logically argue through the operation of the code; and of being able to do this with effort that is not disproportionately large compared to the effort of writing and debugging the code. It is claimed that this approach leads to code which is better understood (hopefully correct), and consequently to far fewer serious bugs and errors, and much less debugging. Dijkstra succintly summed up this approach to program writing in the often-quoted:

"Testing shows the presence, not the absence of bugs."

When you pause to think about it, it is clear that he is correct: no amount of testing can assure us that a large program is in fact completely correct. Your own experience of lurking bugs in compilers, operating systems and manufacturer's software (and perhaps your own software?) should suffice to prove this. To take a simple example however, consider the flow chart of Fig 4. It is not

over-complex as a sub-program, and yet there are ($3^{10} + 3^9 + 3^8 + \ldots\ldots + 3^1 + 1$) = 88573 different control flow paths through this small program! Even without considering the possible data interactions, complete brute force testing of anything but the most miniscule program is not likely.

No; it is essential to use your knowledge of structure in designing and testing programs. Even naive beginners soon learn to do this: for example they assume that if a loop works correctly once it will always work correctly. The natural corollary is however not to test a written program with great effort to ensure that it works correctly, but to write it correctly in the first place and understand it while you are writing it. If then in testing the segment turns out to be incorrect, it is likely to be due either to a clerical error (which is relatively easy to detect, and which compilers can often flag) or to an error in assumption about the program. In the latter case the correction may cure the problem for all time, rather than as so often happens, cure it temporarily by slapping a patch over a mysterious flaw.

How do we go about proving things about programs? The key ideas can perhaps be summed up as follows:

* Programs-in-execution can be divided into two basically different parts: an invariant part which does not change during execution, and which is made up of code, constants, fixed tables, etc., and a part which varies during execution, made up of data, the operating environment, etc.
* If code-like sections of the fixed part have identifiable start and finish points, then it should be possible to make some assumptions about the state of the data-part at the start of the section and to prove from the axioms of the actions and the properties of the code-construct that some consequent assertions hold at the end of the section.
* To prove a program correct, it is necessary to show that the starting assumptions of each code section match up with the assertions of the code section immediately preceding or enclosing it, and that the overall program assertion matches up to the defined problem specification.

Most of us do something like this intuitively. The contribution of structured code is to make us aware of what it is we do, and to encourage the use of code constructs that are well-suited to understanding.

A simple example might make these points clearer. Consider the problem of finding the smallest number in a vector of numbers (or if you prefer, the alphabetically first name in a set of names: the two formulations are not much different). A common method for doing this is to scan each element of the set one-by-one comparing it with a currently smallest known element. This algorithm, which is in some senses an optimal algorithm for the problem, might be coded something like this:

```
 1   PROCEDURE FINDMIN (VECTOR,LENGTH,MINVALUE)
 2      SPECIFY LENGTH, MINVALUE: INTEGER
 3      SPECIFY VECTOR: ARRAY [1 TO LENGTH] OF INTEGERS
 4      DECLARE INDEX, TRIAL: INTEGER
 5      TRIAL ← VECTOR [1]
 6      LOOP FOR INDEX ← 2 TO LENGTH
 7         IF TRIAL < VECTOR [INDEX]
 8            TRIAL ← VECTOR [INDEX]
 9         END IF
10      END LOOP
11      MINVALUE ← TRIAL
12   END PROCEDURE
```

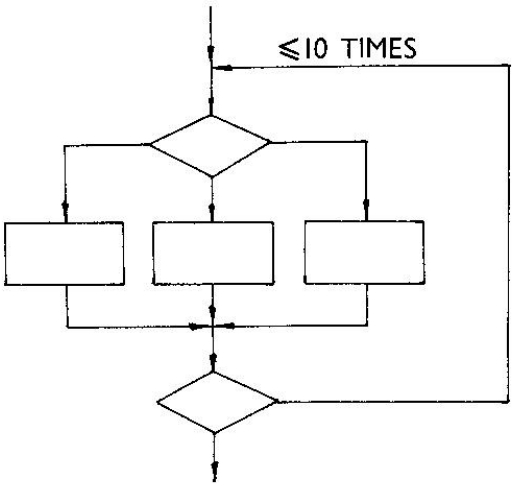

Figure 4 Example of a flow-chart

(Note that the line numbers are there simply for convenience in referring to points in the program.) Lines 1 to 4 are primarily specification code and serve to set up the conditions for the executable code proper. For example the above program is concerned with only integer numbers and has access to non-local objects VECTOR, LENGTH and MINVALUE, as well as the local objects TRIAL and INDEX. Assume that at the program start the variable LENGTH has defined value $\geqslant 1$, and that the array VECTOR also has defined values for all elements from 1 to the value of LENGTH inclusive. No other assumptions are needed. This is then the state of my assertions about the program state just before line 5: whenever we are about to enter line 5 we know such-and-such about LENGTH and VECTOR, and nothing about the values of TRIAL, INDEX and MINVALUE.

It would be pointless to work *sequentially* through the program steps trying to prove this program works, for we would not know what was coming next. Rather we either have to work from the outside in towards the innermost nested constructs, or from the innermost constructs out (or both), knowing or having a fair idea what we are trying to show. In this case I will choose to work outwards and I shall therefore first consider the IF construct (lines 7-9) within the loop. This IF construct alters the variable

TRIAL only. Assume at the start of the IF (before line 6) that TRIAL has some integer value, then the test provides for two cases:

* TRIAL < VECTOR [INDEX], in which case TRIAL takes the same value as VECTOR [INDEX], or
* TRIAL ⩾ VECTOR [INDEX], in which case nothing is done.

Accordingly at the end of the segment TRIAL has a value which is the smaller of that it had before and that of VECTOR (INDEX).

The next enclosing construct is the loop (lines 6-10), for which a quite different proof technique is required (remember that selection requires enumeration of the actions while loops require inductive reasoning). Assume that we know the properties of count-loops, for it would indeed be laborious to prove that every count-loop in fact counted! Then we need to show two things: that each iteration of the loop carries out an action consistent with the target, and that the appropriate initial conditions are right. And it is here that the ingenuity of the programmer comes in for it is not always easy to determine what must be asserted as the action of an iteration. In this case I will **assume** that **before** the j-th iteration (an arbitrary one) the variable TRIAL holds the value which is the least found in all the elements VECTOR [1] to VECTOR [J - 1]: for that is what we want. I can then show that if this is so, then it is also so **after** the j-th iteration, for the action of the body of the loop is to make TRIAL keep the least value of its previous value and the j-th element (see above). Further, and this is important, just before the first iteration TRIAL holds the minimum value of all the elements preceding the first index value (i.e. the value of VECTOR [1]). Therefore since my assumption holds for before the first iteration, it holds after the first and before the second, after the second and before the third ... and so on until the loop terminates. In this case it does so by reaching the element VECTOR [LENGTH] which is supposedly the last element to be considered, and therefore at this point TRIAL will have the value of the smallest element in VECTOR.

And that is all bar the shouting. Line 11 simply passes this value back into the external environment, so that I can assert that at procedure exit both VECTOR and LENGTH are unaltered, and MINVALUE has the desired value. I have hit this trivial program with a sledge-hammer of course, but it is an example which is worth looking over carefully. It illustrates proof techniques applied to selective combination (the IF), and to loops. It also illustrates composition, for notice that in using this procedure in a larger program we need not even know about the internal variables TRIAL and INDEX; indeed INDEX could well have been local to the loop within FINDMIN (it was never used anywhere else). You will realize that the amount of work required to analyse a complete program depends basically upon the number of constructs in the program and their complexity, and not on the total number of paths through the program.

Provided the accuracy of the measure is not taken too seriously, it is possible to construct a control complexity index (CCI) which is a measure of the difficulty of proof of correctness of a program. Assume then that a basic action has a complexity of 1: it takes one unit of effort to comprehend it. The enumerative actions have complexities equal to the sum of the complexities of all sub-actions, while loops have complexity equal to their body complexity plus some extra component due to the reasoning involved. The situation is shown in Fig 7.1. Since none of these measures involve any function growing faster than a sum, the CCI grows approximately linearly with the size of its associated program (not explosively, as normal software managerial experience tends to assert). Further provided that the complexity of any individual component is not too great (such as a very diverse selection) the complexity of a program is localizable: it is possible to work on a fragment ignoring much irrelevant detail. This is the important principle of modulairty shorn of its folklore: that working on one part of a program can be independent of work on another. *Divide and rule!*

9. CONCLUSION

It was the intention of this paper to set a framework for subsequent discussions of the practical consequences of structured programming for industry programmers. In introducing the basic primitives and the use that can be made of such constructs in understanding programs, I hope to have shown that there is perhaps some merit in writing code which is well-structured and in the popular catchphrase **goto**-less. I hope too to have indicated that there is a lot more to structured code (and structured programming) than **goto**-lessness, and that indeed the last word has not been written on the topic. While it would be generally agreed that rats' nest styles of programming are to be abhorred, it is recognized by some that the **go to** and similar constructs do have some role to play in high-level languages in unusual situations.

In concluding, I might remark that it is often very hard to convert existing programmers to a structured way of thought, or to get them to recognize merit in some of these new developments. Two reactions are common: (1) "I've been doing this all along, so I'll read no further. Go teach your grandmother to suck eggs!" and (2) "Yes, but I find that there are too many exceptions...". We are conditioned by our experiences: as Dijkstra is reported to have said, "No one survives early education without some scars. .... It is a requirement for my applicants that they have no knowledge of FORTRAN. ..." In practice this means that even though we may consciously know about other ways of working, our thought-patterns may still be running in obsolete channels, forcing continual translation and non-optimal programming habits. Things are quite otherwise when training newly-hatched programmers: altogether a much easier task since constructions will occur to them (within their paradigm of structured programming) that seldom occur to old-style programmers.

References

BARRON, D.W. (1968): "Recursive Techniques in Programming", *MacDonald Computer Monographs*, pp51-58.

COOPER, J. (1966): "The Equivalence of Certain Computations", *Computer Journal*, Vol. 9, 1, May 1966, pp45-52.

DAHL, O.J., DIJKSTRA, E.W. & HOARE, C.A.R. (1972): "Structured Programming", *Academic Press*, London.

DIJKSTRA, E.W. (1965): "Programming Considered as a Human Activity", *Proceedings of IFIP Congress 65, Spartan Books*, Washington D.C..

DIJKSTRA, E.W. (1968): "Goto Statement Considered Harmful", *Communications of the ACM*, Vol. 11, 3, March 1968, pp147-148.

DIJKSTRA, E.W. (1968): "A Constructive Approach to the Problem of Program Correctness", *BIT*, Vol. 8, 2, pp174-186.

DIJKSTRA, E.W. (1969): "Notes on Structured Programming", *Technische Hogeschool Eindhoven*.

DIJKSTRA, E.W. (1970): "Structured Programming", in BUXTON & RANDELL (ed) (1970): "Software Engineering Techniques", *NATO Science Committee*.

DIJKSTRA, E.W. (1972): "The Humble Programmer", *Communications of the ACM*, Vol. 15, 10, October 1972, pp859-866 (ACM Turing Aware Lecture).
GRIES, D. (1974): "On Structured Programming – A Reply to Smoliar", *Communications of the ACM*, Vol. 17, 11, November 1974, pp655-657.
HOARE, C.A.R. (1969): "An Axiomatic Approach to Computer Programming", *Communications of the ACM*, Vol. 12, 10, pp576-580 & p583.
HOARE, C.A.R. (1972): "High-level Programming Languages – The Way Behind", in "High-level Languages – The Way Ahead", *National Computing Centre Publications*, Manchester.
HOARE, C.A.R. (1973): "Recursive Data Structures", *Computer Science Department CS-73-400*, Stanford University.
HOARE, C.A.R. (1973): "Hints on Programming Language Design", *Computer Science Department CS-73-403*, Stanford University.
HOARE, C.A.R. (1973): "Proof of a Structured Program: The Sieve of Eratosthanes", *Computer Journal*, Vol. 15, 4, November 1972, pp325.
HOARE, C.A.R. & WIRTH, N. (1972): "An Axiomatic Definition of the Programming Language Pascal", *Berichte der Fachgruppe Computer-Wissenschaften 6*, Eidgenossische Technische Hochschule, Zurich.
KNUTH, D.E. (1973): "A Review of 'Structured Programming' ", *Computer Science Department CS-371*, Stanford University.
KNUTH, D.E. (1974): "Structured Programming with Goto Statements", *Computer Science Department CS-74-416*, Stanford University.
KNUTH, D.E. (1969): "The Art of Computer Programming – Volume 2: Seminumerical Algorithms", *Addison-Wesley*, pp195-207.
KUHN, T.S. (1970): "The Structure of Scientific Revolutions", 2nd edition, *The University of Chicago Press.*
LECARME, O. (1974): "Structured Programming, Programming Teaching and the Language Pascal", *SIGCSE Bulletin*, Vol. 6, 2, June 1974 pp9-15, (also in *SIGPLAN Notices*, Vol. 9, 7, July 1974, pp15-21.
McKEEMAN, W.M. (1964): "Algorithm 239 – Free Field Read", *Communications of the ACM*, Vol. 7, 8, August 1964, pp481-482.
MILLS, H. (1973): "Mathematical Foundations of Structured Programming", *IBM Technical Report.*
PETERSON, W.W., KASAMI, T. & TOKURA, N. (1973): "On the Capabilities of While, Repeat and Exit Statements", *Communications of the ACM*, Vol. 16, 8, August 1973, pp503-512.
WAITE, W.M. (1970): "Building a Mobile Programming System",*Computer Journal*, Vol. 13, 1, February 1970, pp28-31.
WAITE, W.M. (1970): "The Mobile Programming System – STAGE 2", *Communications of the ACM*, Vol. 13, 7, July 1970, pp415-421.
WIRTH, N. (1971): "The Design of a Pascal Compiler", *Software Practice and Experience*, Vol. 1, 4, pp309-333.
WIRTH, N. (1972): "The Programming Language Pascal (Revised Report)", Berichte der Fachgruppe Computer-Wissenschaften 5, Eidgenossische Hochschule, Aurich.
WIRTH, N. (1972): "On Pascal, Code Generation and the CDC 6600 Computer", *Computer Science Department CS-72-257*, Stanford University.
WIRTH, N. (1973): "Systematic Programming: An Introduction", *Prentice-Hall*, Englewood Cliffs.

Appendix: On Mutual Recursion

Consider the following two procedures, here written to conform to the syntax of Algol 60 to minimize misunderstandings:

```
integer procedure factorialа(n);
    integer n; value n;
    factorialа:= (if n = 0 then 1 else factorialа(n – 1)
    x n);
integer procedure factorialb(n);
    integer n; value n;
    factorialb:= (if n = 5 then 120 else factorialb(n +
    1) ÷ n);
```

Both procedures are asserted to compute the factorial function of n (n! = n x(n – 1) x (n – 2) x ... x 2 x 1) for values of n lying in the range 0 to 5 inclusive. No other values of n will ever be presented. A little thought will convince you that both procedures are in fact correct (they terminate and do so with the correct value), differing only in efficiency. The procedure factorialа is faster if small values of n predominate, while factorialb may be preferable if values near 5 predominate.

Suppose now that we make some minor changes. We know that factorialа and factorialb compute the same thing, so suppose we alter the executable body of factorialb to read:

```
    factorialb:= (if n = 5 then 120 else factorialа (n + 1) ÷
    n);
```

then do the two procedures still correctly compute the required values? Further if we also make the corresponding change in factorialа to:

```
    factorialа:= (if n = 0 then 1 else factorialb (n – 1) x
    n);
```

what then? If we make the assumption that factorialb is correct then it follows that factorialа is too; conversely assuming that factorialа is correct then factorialb must be. And yet if only one of the lines are changed both procedures are correct, but if both are changed the whole thing falls apart (and in this constructed example fails to terminate). Where has it all gone wrong? Essentially by changing the two procedures to be mutually recursive we have created a circular argument, and to show that either of the procedures is correct it becomes necessary to consider *both together*. In this very simple example the error is of course in not ensuring that the mutual calls result in some consistent progress towards termination. Much more elusive errors are possible.