STRUCTURED PROGRAMMING FOR REALTIME COMPUTER SYSTEMS

UCC TR 74-2
November 15, 1974

by
P. C. Patton
W. R. Franta

CONTENTS

1.	Structured Programming	1
	1.1 The search for a definition	1
	1.2 The great goto controversy	3
	1.3 Research frontiers in structured progamming	8
2.	Application to Tactical Data Processing	10
	2.1 Background and special problems	10
	2.2 Some efficiency problems	10
	2.3 Some efficiency examples	12
3.	Disciplined Use of the GOTO Statement	14
	3.1 Some alternatives	14
	3.2 Necessary use of the goto statement	15
	3.3 The restrained goto: some programming conventions	18
4.	Structuring a Tactical Module: BVEQJ	20
	4.1 The computer case statement	20
	4.2 A case flow analysis of BVEQJ	22
	4.3 Comparisons and cost analysis	30
5.	Conclusion	32
6.	An Annotated Bibliography on Structured Programming	34
	Ashcroft	35
	Baker-Burstall	36-38
	Chanon-Courtois	39-40
	Denning-Dijkstra	41-43
	Fisher-Franta	44
	Gilbert-Gries	45
	Habermann-Hopkins	46-50
	Kernighan-Knuth	51-53
	Leavenworth-London	54-55
	Miller-Mills	56-57
	Nassi-Naur	58
	Parnas-Plum	59-60
	Schwartz-Smoliar	61
	Tenny-Tsichritzis	62
	Weizenbaum-Wulf	63-65
	Yourdon	66
	Zahn-Zelkowitz	67
	Supplementary Listing	68-70

FIGURES

1.1	Basic Structural Schemas	5
4.1	The Structure of BVEQJ in CMS-2 Like Notation	23-24
4.2	Flowchart of BVEQJ by Statement Numbers of Figure 4.1	25
4.3	BVEQJ Written in Multilevel Case Statement Notation	27
4.4	Flow Diagram of the BVEQJ Version in Figure 4.2. Repeated Code Outlined in Dashed Line Blocks.	28
4.5	Improved Version of BVEQJ in Indented Algol-like Notation	29
4.6	Structured Program Costs for BVEQJ	31

1. STRUCTURED PROGRAMMING

A crisis has been developing in the computer programming field over the past decade. Even those who clearly saw it coming have been able to do but little to arrest the seriousness of it. The crisis centers around the negative economy of scale associated with programs for large data systems. As computers have increased in size, speed and capability generally, they have been applied to even more complex yet even more realistic, relevant problems. Engineering technology has somehow been just able to keep abreast of the problem caused by these increases in hardware complexity, but programming technology has been falling behind. The cost per instruction of a large program is usually higher than that of a smaller one, the reliability is relatively lower and the maintainability is so poor that such monoliths are often deliberately redesigned and reprogrammed from scratch rather than repaired or modified. Structured programming is an attempt from within the programming profession itself to place the art on a somewhat more scientific basis, in the sense of getting predictable results, repeatable results and extendability of experience.

1.1 THE SEARCH FOR A DEFINITION

Unfortunately a great controversy has grown up around structured programming, and this is primarily due to the term's lack of a clear definition. Many of the important concepts in computing have yet to be defined precisely (e.g. "throughput," "real-time," "transaction") yet are used intuitively by most computer people according to consensus. In many ways computer technology plays the role in our society that theology played in the middle ages. We hesitate to rush to a precise definition, which like a credal statement will split the faith into two camps of believers. Denning has addressed the definition problem in an articulate comment, (D1) and Gries has summarized the situation up to November 1974. He lists the following set of impressions as those most people seem to have of structured programming (G3). Taken together these impressions form a comprehensive, if not rigorously consistent, view of the subject. The first five were quoted in Denning's letter (D1).

1. It is a return to common sense.
2. It is a general method which leading programmers use.
3. It is programming without use of the goto statement.
4. It is the process of controlling the number of interactions between a local task and its environment so that the number of interactions is some linear function of the parameters of the task.
5. It is top-down programming.
6. It deals with converting large, complex flowcharts into standard forms which can be represented by iterating and nesting a small number of basic control logic structures (G3, M6).
7. It is a manner of organizing and coding programs that makes them easily understood and modified (G3, D9).
8. It controls program complexing through theory and discipline (G3, M2).
9. It should be characterized not by the absence of goto's but by the presence of structure (G3, M2).
10. It has a major value in keeping a program correctness proof feasable (G3, D2).
11. It takes a correctness proof as fundamental (G3).
12. It allows verification of the correctness of all steps in the design process leading to a self-explaining and self-defensive programming style (G3).
13. It is no panacea but really a formal notation for orderly thinking and a discipline which must be acquired and continually reinforced through conscious effort (G3).

It is true that structured programming is all of these things weighted one relative to another depending on the program environment or application, but they do not form a convenient definition. Grics quotes Hoare to give a shorter, more intuitive definition:

> The task of organizing one's thought in a way that leads, in a reasonable time, to an understandable expression of a computing task, has come to be called structured programming.
> (Quoted from G3.)

The choices for a definition at this state of development would appear to be either enumerative but inconcistent, intuitive but imprecise, or none at all. Denning in a recent epistle (D11) has sounded the latter sour note:

> "On the basis of all this information,* I have come to the conviction that my secret wish for a definitive treatment of 'structured programming' in unachievable: there is no fixed set of rules according to which clear, understandable, and provable programs can be constructed. There are guidelines, of course, and good ones at that; but the individual programmer's style (or lack of it), his clarity of thought (or lack of it), his creativity (or lack of it), will all contribute significantly to the outcome. [author's underlining]

Denning also noted that during the process of assembling this issue he saw:

> "examples of programs which looked unstructured (containing, for example, goto statements), but which were quite understandable. [He also] saw examples of programs which looked well indented and properly commented, but on closer examination turned out to be unnecessarily complex by an order of magnitude." [author's underlining]

He closes this letter with the comment:

> "When Dijkstra and others first used the term "structured programming" they clearly intended it in the widest sense, encompassing notions of control structure (exemplified by the restricted forms of statements), data structure (exemplified by data objects manipulated only by operations to which the structure itself was private), proper documentation (exemplified by the advice to evolve a program structure characterized by a sequence of versions, each being a refinement of the previous one), and proper control of the interface with the world (examplified by the careful use of input/output statements and verification of data). Now the term has been barbarized and oversimplified. 'Structured programming' now seems to mean but a component of what structured programming was originally envisioned to be. [author's underlining]

1.2 THE GREAT GOTO CONTROVERSY

Much of the controversy over structured programming has been between proponents of items three and nine as listed above. Few protagonists will dispute the fact that the haphazard use of goto's leads to unmanageable programs. Such programs are complex and difficult to understand because their "connectivity" is too high. Structured programs are on the other hand based on the Structure Theorem (B5, M2) which states that any

---

* Papers he reviewed for a special issue of Computer Surveys on (structured) programming.

proper program (a program with one entry and one exit) is equivalent to a program that contains as logic structures only:

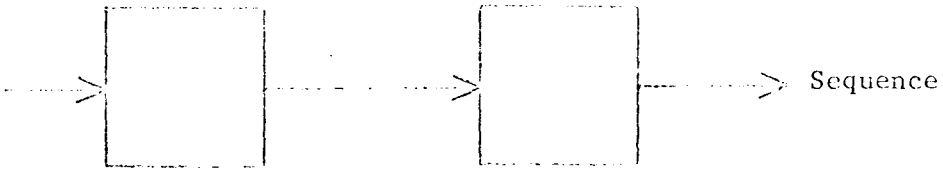
- a sequence of two or more operations
- a conditional branch to one of two operations and return (if a then b else c)
- a repetition of an operation while a condition is true (do while)

Each of the three schemas itself represents a proper program (see Figure 1.1). A large and complex program may then be developed by the appropriate nesting of these three basic schemas within each other. The logic flow of such a program always proceeds from the beginning to the end without arbitrary branching. Where only these structures are used in the programming, there are no unconditional branches or statement labels to which to branch.

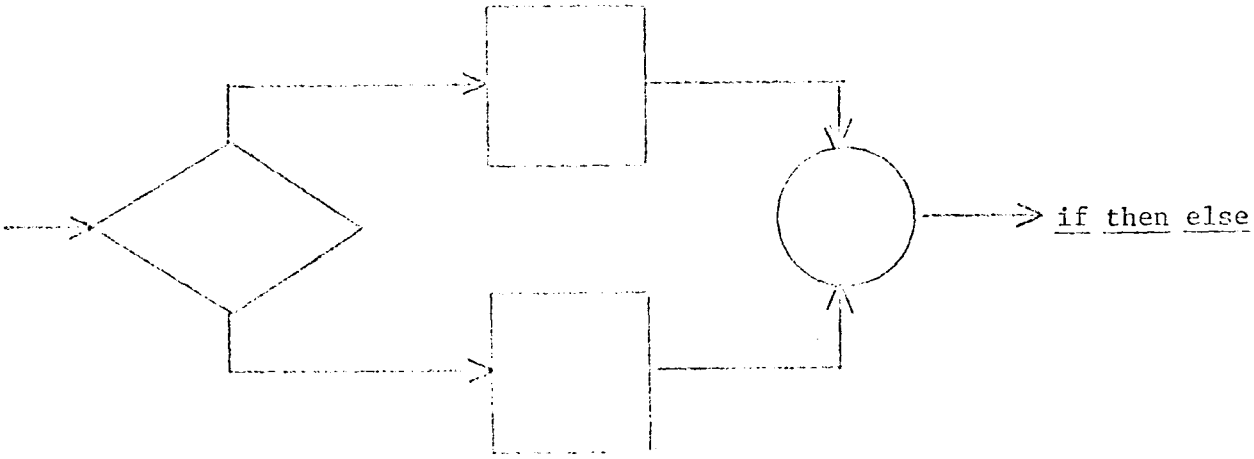
Structured programming reduces the arrangement of the program logic to a process like that found in engineering where logic circuits are constructed from a basic set of schemas. As such, it represents a standard based on a solid theoretical foundation and does not require ad hoc justification, case by case, in actual practice (M5).

Several conventions are included as a supporting part of the technique. For example, strict attention is paid to the indentation of the logic structures on the printed page so that logical relationships in the coding correspond to physical position on the listing. Thus, a pictorial representation of the logic is gained from the indentation. Another convention is that of segmenting code into reasonable amounts of logic each of which is easily understandable. Each segment of code, whose internal operations may be any combination of the basic logic structures, must itself represent one of the basic logic structures. Thus, each code segment becomes a logical entity to be analyzed, coded and read at one time.

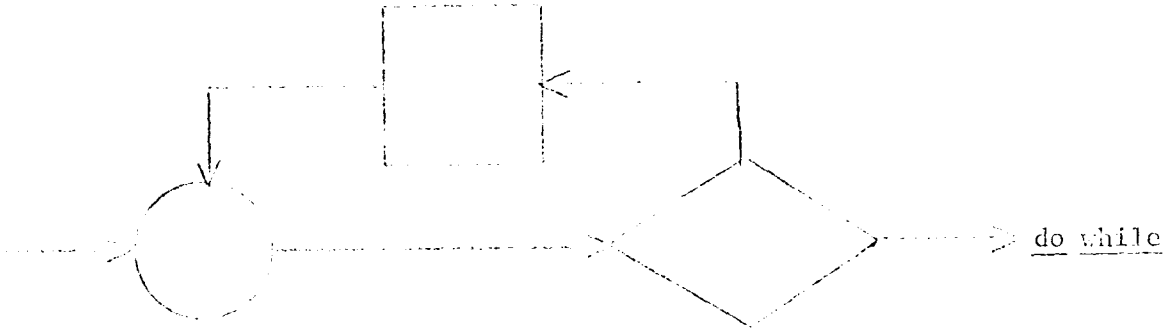
High level languages can be made to a large extent self-documenting. For assembler level languages, macros will provide the basic logic structures, giving these languages the readability and self-documenting attributes of higher level languages. The use of the basic logic structures coupled with indentation and segmentation rules, makes unnecessary the preparation of flowcharts. Simple extensions of the



Sequence of two operations



Conditional branch to one of two operations and return



Operation repeated while a condition is true

FIGURE 1.1 BASIC STRUCTURAL SCHEMAS

three basic logic structures are allowed. These do not affect the spirit of structured programming, but may result in more efficient use of computer time and storage.

By definition, a goto-less flowchart is susceptable to a sequence of transformations which reduce it to a single process box. One can imagine such a sequence in which: (1) the correctness of the replaced construct has been verified, and (2) the new process box contains a more macroscopic description of what the replaced portion does. This sequence is able to form both a proof of the original program's correctness as well as documentation of what it does (W8). This feature also allows use of the top-down technques during program development.

Traditional software development has evolved as a bottom up procedure whereby the lowest level processing programs are coded first, unit tested, and made ready for integration. Additional code in the form of driver programs is needed to perform the unit testing and lower levels of integration testing. Data definitions and interfaces tend to be simultaneously defined by more than one person and often in inconsistent ways. During integration these definition problems are recognized and integration is delayed while the data definitions and interfaces are correctly defined and the processing programs are reworked (and unit tested again) to accommodate the changes. It is often difficult to isolate a problem during the traditional system integration cycle because of the large number of possible sources. Management control often is ineffective during much of the traditional development cycle because there is no coherent, visible product until system integration is completed.

The top down approach is patterned after the natural approach to system design and requires that programming proceed from developing the control architecture or interface statements and initial data definitions downward to developing and integrating the functional units. Top down programming is an ordering of system development which allows for continual integration of the system parts as they are developed and provides for interfaces prior to the parts being developed.

In top down, structured programming, the system is organized into a tree structure of segments. The top segment contains the highest level of control logic and decisions within the program, and either passes

control to lower level segments, or identified lower level segments for in-line inclusion. This process continues for as many levels as required until all functions within a system are defined in executable code.

Many system interfaces occur through the data base definition in addition to calling sequence parameters. The top-down approach requires that the data base definition statements be coded and that actual data records be generated before exercising any segment which references them.

This approach provides the ability to evolve the product in a manner that maintains the characteristic of being always operable, extremely modular, and always available for successive levels of testing that accompany the corresponding levels of implementation. The quality of a system produced using the approach is increased, as reflected in fewer errors in the coding process. The act of structuring the logic calls for more forethought, and the uniformity and single entry, single exit attribute of the structured code itself contribute to the reduction in errors (M5).

There is a great deal of literature for and against the use of goto statements and hopefully all of it is referenced in the annotated bibliography of the report (see especially H15, K1, K2, K3, L1, W8, W10). To the tactical programmer or any other programmer engaged in on-line, real-time programming, the central issue is not one of program esthetics, program managability, or even program "morality," but rather one of efficiency. Knuth comments that "it seems that fanatical advocates of the New Programming are going overboard in their strict enforcement of morality and purity in programs. Sooner or later people are going to find that their beautifully structured programs are running at one half the speed or worse of the dirty old programs they used to write, and they will mistakenly blame the structure (instead of the real culprit, the system overhead caused by typical compiler implementation of Boolean variables and procedure calls)" (K3).

In the same document Knuth summarized a discussion of goto elimination with the comment:

> In other words, we shouldn't merely remove goto statements because it's the thing to do, like taking vitamin C or vitamin E; the presence or absence of goto statements is not really the issue. The underlying structure of the program is what

counts, and good structure can be expressed in FORTRAN or COBOL or even assembly language. The goal of goto elimination is not the elimination, rather the reformation so that only straightforward control structures remain. Once the structure is there, it is also extremely valuable to make it readily visible to the eye, by using syntactic aids like nesting and indentation; this important notational advantage is what high-level programming languages (especially those of the future) will give us, but in a sense this is only of secondary importance while the existence of good structure is the primary goal. Dijkstra's article (D2) asks for more new language features to replace the goto, structures which encourage clear thinking, instead of the goto's temptations to make complications.

1.3 RESEARCH FRONTIERS IN STRUCTURED PROGRAMMING

A good way to present any area of scientific endeavor and particularly a relatively new one, is to outline its major unsolved problems or its research directions. At least one well informed author has categorized these directions for structured programming as programming methodology, program notation, program correctness and program verification (G3).

The goal of research in programming methodology is to devise orderly, efficient methods for developing readable, correct programs, to identify and explain tools and techniques for solving programming problems and to find out how to think clearly when programming. Some of the current concepts of interest here are: "levels of abstraction," "step-wise refinement," "top-down programming," "solve a simpler problem first," and "find a related problem." It should be clear that one technique alone will never suffice (for example, top-down programming). A programmer needs a collection of methods and techniques for attacking a problem.

A cluttered programming language can hinder the user from thinking clearly; a restricted language can hide the best algorithm. Research in this area of structured programming is devoted to improving notation: by looking for better notation and language features which can help simplify the process; by determining which control structures are best suited for describing algorithms correctly and clearly; and by illustrating how to describe data structures in more appropriate ways.

So far the programmer has tended to put too little emphasis on initial correctness and too much emphasis on debugging. But debugging can never show the absence of errors, only their presence (D2). Current methodology

in structured programming indicated that one should try to develop a program and its proof of correctness hand-in-hand. Unfortunately, most of the work on correctness proofs has been quite formal and mathematical and has yet been of no use to the programmer. But some important practical ideas have arisen in the research on axiomatic approaches to programming language definition. One such idea is that we should not look so much at how a program changes values of variables, but instead of how relations among the variables remain the same (G3). The most significant advance is Hoare's invariant relation axiom for while loops; this axiom gives a practical approach to understanding iteration; it bridges the gap between the static aspect of a loop (how it is read) and its dynamic aspect (how it gets executed) (G3).

Program verification is the process of certifying a program to be correct whether or not it has been honored by a correctness proof. Many published program proofs have turned out to be incorrect (independent of the program's correctness), thus certification or "debugging" is always a requirement. Some research is being done in this area (SH1), but it does not seem to be a primary research direction at the present time (G3).

2. APPLICATION TO TACTICAL DATA PROCESSING

2.1 BACKGROUND AND SPECIAL PROBLEMS

Tactical programs have always been developed in a background of tension between two categories of goals. On the one hand have been the nearer goals of efficiency in the use of machine resources, i.e., computer time, memory space, sensor accuracy, etc. On the other hand were the farther away goals of program manage-bility, i.e., development lead-time, modularity, maintainability, expandability, etc. There seemed to be no middle ground between those factors of microscopic cost/effectiveness and macroscopic cost/effeciencies. Efforts at software engineering to emulate hardware engineering in the sense McHenry (H5) and many others have called for it has failed so far to develop a full-blowm disciplinary approach. Structured programming promises to develop into a software engineering discipline if programmers can steer the course between over-challenging it in its infancy and following after it willy-nilly as the latest fad.

The realtime programmer has a special requirement on his work in the economy of memory space and processor time which he must observe if his work is to perform up to specification. This characteristic of his work tends to make him reluctant to accept programming niceties such as high level languages a decade ago or goto-less programming now. His counter-parts in the commercial and scientific programming fields tend to follow such novelties more readily, since they are less constrained. Some critics have tried to make realtime programming a difference of degree rather than kind; but as Dijkstra points out in this regard, a difference of degree encompassing several orders of magnitude is already a difference of kind (D2).

2.2 SOME EFFICIENCY PROBLEMS

A study typical of the near goal evaluation mentioned above is the CMS-2Y Compiler Code Generation Study done on assembly level code generated for the AN/BQQ-5 sonar program. Although this sort of event driven program is somewhat an extreme case, it certainly illustrates the problem of universal application of dogmatic programming rules to

all application areas without regard to natural constraints. Although the CMS-2Y compiler language does not have available all of the structured code modules described above, it is a relatively efficient compiler. Thus one cannot blame all of the problems that arise on this particular compiler; they result from the attempt to structure or its side effects (code duplication and time lost in procedure calls), which are typical of virtually all present day compiler generated programs.

For the sake of completeness a brief review of the results of this study (IBM Electronic Systems Center, No. 74-PY3-001) will be given here. Three cases were analyzed. The first was a time critical segment of code from the passive broad band processing program. The CMS-2Y code was developed from the assembler language listing, a fact which likely introduced some bias in the result but surely does not render it invalid. Within the limits of this decompilation technique and the small segment size of the code, an effort was made to create a top-down and structured code. The author's note essential success, with some goto's. The structured CMS-2Y code (without tables) showed a 46.3% increase in space and a 70.5% increase in time. An effort to replace compiler generated code with careful hand coding was able to bring these figures down to 25.1% and 30.4% respectively. Although more efficiency could probably have been gained by further direct code substitution it would have been at the loss of program structuring.

The second case was a control program which was recoded in CMS-2Y without an attempt at top-down and structured programming but resulted in space and time increments of 61% and 41.5%, with hand optimizing to yield 38.4% and 19.8%. The third case is of greater interest since the code was developed from a top-down/structured design. Rather than comparing the compiler code to existing AN/BQQ-5 assembly code, it was compared to a total direct code implementation. Two different such design approaches were taken, but the best showed a 34% space handicap over direct code and a 39% time handicap. Hand optimization was able to reduce these to 0% and 16.4%.

The conclusion made in this report was that utilization of the CMS-2Y compiler as the sole sonar implementation language does not appear to be viable. Thus direct code or assembler language will be used for the time

and core critical areas of the system and the compiler used in the remaining areas. This conclusion is good as far as it goes, but a broader perspective of the situation may allow one to enhance it a bit. The experiments quoted from the report show that hand optimizing is most effective, i.e., yields the greatest gains, when done on a program resulting from a top-down and structured design process. If a program as tricky as this sonar processor can be designed top-down and structured and then hand optimized to show no handicap in space utilization and only 16.4% time cost, then there is indeed hope for structured programming in the tactical data processing application environment!

The technique needed will turn out to be that called for by Knuth and some of the cooler heads in the controversy, i.e., structured programming with goto's.

2.3 SOME EFFICIENCY EXAMPLES

An attempt to obtain a copy of the AN/BQQ-5 sonar program for illustrative purposes was not successful, thus some classic examples used by Knuth and others will be reported here to underline the case for restricted use of the goto locally within the overall globally structured program. These examples will also show some cause for the sort of time penalties structured programming brought to the sonar program fragments discussed above.

The basic example is that of searching a linear array A[m] for x (K2, K3).

```
I for i: = 1 step 1 until m do
          if A[i] = x then goto found;
          not found; m:=i:=m+1; A[i]:=X;B[i]=0;
          found: B[i]:=B[i]+1
```

The data structure can be modified to eliminate the goto.

```
II   A[m+1]:=x; i:=1;
     while A[i] ≠ x do i:= i+1;
     if i>m then m:=i; B[i]:=1 else B[i]:=B[i]+1;
```

Example II also makes the inner loop faster. If these were codes in assembly language and the values of i and x kept in registers, program

I will take 7n + 7 memory references and program II will take 4n + 11. If they were both compiled on a good compiler with bounds checking suppressed, the corresponding run times would be proportional to 16n + 5 and 13n + 18. In both cases elimination of the goto saves time, either 43% or 19% but it should be noted that the cost of using a higher level language outweighs the gain. If this example were really a time critical inner loop we would recode it as follows for efficiency:

```
III  A[m+1]:=X;i:=1; goto test;
     loop:  i:= i + 2;
     test:  if A[i]=x then goto found;
            if A[i + 1] ≠ x then goto loop;

            i:=i + 1;
     found:  if i>m then m:=i;
             B[i]:=1 else B[i]:=B[i]+1;
```

The running time is now proportional to 3.5n + 11.5, a significant gain and yet it has been done by the introduction of only three gotos! It is even beginning to look like a piece of radar or sonar data processing code.

3. DISCIPLINED USE OF THE GOTO STATEMENT

3.1 SOME ALTERNATIVES

Goto statements can often be eliminated by the use of Boolean variables, but some compilers deal so inefficiently with Boolean variables that one is reluctant to test them in an inner loop. Ashcroft and Manna have found a nice way to remove goto's from any program by introducing Boolean variables and replicating the code yet preserving much of the program's flowchart topology. The existence of their construction shows that goto elimination can be applied to badly structured programs and the resulting programs will be just as badly structured (D11).

Zahn has introduced an event indicator which allows a new form of iteration clause: (Z1, K3)

```
loop until <event>_1 or <event>_m:
        <statement list>0; repeat;
then    <event>_1→<statement list>_1;
        .
        .
        .
        <event>_n→<statement list>_n;
```

and uses a new statement <event> to signal that the desired event has occurred. This statement is allowed only in the scope of the loop which declares that event. The body of the loop (<statement list>$_0$) is executed repeatedly until one of the named events occurs, then the corresponding statement list is executed.

Knuth points out that this use of an event statement is semantically equivalent to a restricted form of the goto statement (i.e., with a very local scope), which Landin discussed in 1965 (K3, L6). He declared labels at the beginning of each block, just as procedures are done with each having a <label body> just as a procedure has a <procedure body>. Within the block whose heading contains a declaration of label L, the statement goto L means "execute the body of L, then leave the block" (C3, L6). It is probably clearer for the programmer to specify events (Zahn's method) or Bochman's (B4) than labels (Landin's method).

During the last few years languages have appeared in which the designers proudly announced that they have abolished the goto statement. Perhaps the most prominent of these is BLISS (W9), which originally replaced goto's by eight so-called "escape" statements. But the authors of (W9) say, "Our mistake was in assuming that there is no need for a label once the goto is removed," and they later added a new statement, "leave (label) with (expression)" which goes to the place after the statement identified by (label). Other goto-less languages for systems programming have similarly introduced other statements which provide "equally powerful" alternative ways to jump. In other words, it seems that there is widespread agreement that goto statements are harmful, yet programmers and language designers still feel the need for some euphemism that "goes to" without saying goto.

The interested reader is directed to reference K3, C3, B4, L6 for further details and examples of these techniques, for the purposes of this study it is sufficient to point out that they are all semantically equivalent to a goto with restricted scope.

3.2 NECESSARY USE OF THE GOTO STATEMENT

With respect to current languages which are in wide use such as COBOL, FORTRAN, and CMS-2, there is the practical consideration that the goto statement is necessary. Even where a language is reasonably well suited to programming without goto, the elimination of this construct may be at once too loose and too restrictive (M15). PL/I provides some interesting examples here. One exits from an Algol procedure when the flow of control reaches the end bracket. PL/I provides an additional mechanism, an explicit return statement. In many cases elimination of goto statements can be done only by adding exits, thus there are no goto statements, but two return statements cause an exit from both the procedure and the iterative do. Thus the procedure has control structures which have more than one exit and one-in-one-out control structures were a principal reason for avoiding goto. Should the PL/I programmer add a rule forbidding return? He can program without the return, too, but this involves the introduction of a new variable, switch, and a new test. If one assumes that the introduction of gratuitous identifiers and tests is undesirable perhaps return is a desirable construct even though it can result in multiple exit

control structures. Actually, procedures with several return statements are not necessarily difficult to read and modify because they follow the top to bottom pattern and maintain the obvious predecessor characteristic, while avoiding the introduction of new variables. Return is therefore preferable to the alternative of introducing new variables and tests (M15).

However, return is a very specialized statement. It only permits an exit from one level of one type of control, the procedure. One could generalize the construct to apply to multiple levels of control and to do groups or begin blocks as well as procedures. This is like the Bliss leave construct above. Lacking such language, the PL/I user must content himself with goto. The good programmer, who understands the potential complexity which results from excessive use of goto, will attempt to recast his algorithm. Failing to find an elegant restatement, he will insert the label and its associated goto out of the desired control structure. The label stands there as a warning to the reader of the routine that this is a procedure with more than the usual complexity. Also, the label point catches the eye. It is immediately apparent when looking at this statement that it has an unusual predecessor. The careful reader will want to consult a cross reference listing to determine the potential flow of control.

A rewrite of the procedure or restructuring of the data may be in order. But if that fails one may be driven to a rewrite in assembly language. There is an intermediate alternative which may solve the problem without resort to an assembler. The programmer who writes structured programs uses certain techniques such as the introduction of procedures and the repetition of code which can result in the loss of time and space. Given the idiosyncracies of many compilers, a little reorganization of code and a few goto statements inserted by a clever programmer can often improve performance. One should give up a structured program in a higher level language only after performance bottlenecks have been clearly identified and then only give up what is absolutely necessary. A slightly contorted procedure in a higher level language may be an attractive alternative to one written in assembly language. The villain here is the compiler which produces bad code in some situations. Elimination of the goto would not dramatically ease the problems of compiler writers. Even in compilers which do extensive

control flow analysis, a small percentage of implementation effort is devoted to that task. A more interesting subject for compiler writers is the identification of those optimizations which improve the performance of programs written with none or very few goto statements. Viewed in this light the existence of well structured programs imposes an additional obligation and more work on compiler writers. This is work which they should eagerly accept so that programmers will not have to make the trade off between a well structured program and one that performs well. More work is required in this area.

3.3 THE RESTRAINED GOTO; SOME PROGRAMMING CONVENTIONS

Thus far the argument has progressed to the point that it is clear that goto's can be eliminated but the cost of doing so in online realtime programs may be too high given the capabilities of current day compilers. Although many new features have been suggested, the majority of them are euphemisms for the goto statement since they are really disguised gotos, restricted to a local segment of code. It is not necessary to wait until these language features are available; one can code in a high level language from a top-down/structured design such that he produces a program that is globally structured but locally optimum. That is, he may employ goto statements carefully with restricted scope and even insert direct code when absolutely required for efficiency. Some programming conventions from the literature are quoted here as guidelines for such usage.

1) Certain goto statements which arise in connection with well-understood transformations are acceptable, provided that the program documentation explains what the transformation was (K3).

This situation is very similar to what people have commonly encountered when proving a program correct. To demonstrate the validity of a typical program Q, it is usually simplest and best to prove that some rather simple but less efficient program P is correct and then to prove that P can be transfored into Q by a sequence of valid optimizations. Knuth says that a similar thing should be considered standard practice for all but the simplest software programs: A programmer should create a program P which is readily understood and well-documented, and then he should optimize it into a program Q which is very efficient. Program Q may contain goto statements and other low-level features, but the transformation from P to Q should be accomplished by completely reliable and well-documented, 'mechanical' operations.

2) If the last action of procedure A before it returns is to call procedure B, simply goto the beginning of procedure B instead (K3).

It is easy to confirm the validity of this rule, if for simplicity we assume parameterless procedures. For the operation of calling B is

to put a return address on the stack, then to execute B, then to resume A at the return address specified, then to resume the caller of A. The above simplification makes B resume the caller of A. When B=A the argument is perhaps a little tricky, but it's all right.

3) Modularize. Use subroutines whenever possible. (K1)

Excessive use of labels (or statement numbers) and gotos is often the hallmark of undisciplined design. Tracing flow of control can be next to impossible if there are too many potential paths from one point to another. Even when such code is correct, it is hard to understand and thus even harder to modify.

4) Avoid gotos completely if you can keep the program readable. (K1)

One way to gain a real appreciation for proper control structures is to look at a large program that uses almost none of them.

5) Use gotos to implement a fundamental structure. (K1)

Some programmers argue that the goto statement should be eliminated completely. This is not usually possible in Fortran or convenient in PL/I; but if one can identify the role of each goto in implementing the basic structures the result is clear. Programs can contain gotos if they are used in a stylized, disciplined way, implementing some basic program structure.

6) The lack of a case statement in language is a clear deficiency.

The resourceful programmer can contruct one out of a goto. This does not make up for the lack of a case statement, but it does point up an interesting and highly legitimate use of goto. Imaginative programmers will, from time to time, develop new control constructs as Hoare invented the case statement. Those that are worthwhile will be informally defined and implemented with a macro preprocessor. The better ones will appear in experimental compilers and eventually the best will find their way into the standard languages. Such inventions are often very hard to implement with macro preprocessors for existing languages without use of the goto construct. There is still room for the incorporation of unusual control mechanisms into existing block structure languages. Decision tables are a prime example (H15).

4. STRUCTURING A TACTICAL MODULE: BVEQJ

Franta and Maly have suggested the computed case statement as a program control structure (F4). Their notation was tried on a typical goto infested piece of tactical code to see if some structure could be brought to the eye from what appeared at first to be a rat's nest of program control paths. Once structure was apparent, then the cost of this structure in running time and storage space could be estimated. The analysis given here should not be considered as a criticism of the code that was restructured; it should be kept in mind that the programmer's design goals were probably something like "write and debug as soon as possible a program to do process BVEQJ in the minimum CPU time using the minimum core." The programmer may have met these goals but sacrificed readibility and convenient maintainability on his way to them. An exposition of the computed case statement will be followed by its application to BVEQJ and a discussion of the conclusions that can be made from this brief analysis.

4.1 THE COMPUTED CASE STATEMENT

The computed case statement of Franta and Maly can be understood as a variation of both the case and if-then-else statements. It has the following form:

```
case flow
    <decision tree>
of   1:  S1;
     2:  S2;
      .
      .
      .
     n:  Sn;
end;
```

The syntax of the decision tree is best described by the tree grammar below. A tree grammar is similar to BNF except that in a production a symbol followed by a list of symbols is to be interpreted as a description of a root node together with its immediate descendants.

<decision tree>:=<predicate>(<true part>,<false part<)

<true part>: = <decision tree>|<*case index>

<false part>: = <decision tree>|<*case index>

<predicate>: = <statement<:<boolean expression>?|<boolean expression>?

The semantics of this construct is more or less implied by the naming of the symbols. That is, the predicate is evaluated and if the terminating Boolean expression has the value true the left descendant, <true part>, is traversed otherwise the right descendant is traversed. The traversal is completed when a leaf designating the case index is reached. Thus, each traversal results in a selection of a base index, j, and thereafter in the execution of the corresponding case statement, S_j. The number 1,...,n appearing in the schema above are not labels but rather case designators; hence, they may not be referenced outside the decision tree.

An example of the use of this statement is seen in the following program:

```
v:  = g(x,y);
x:  = f(x,y);
case flow        x<v?
          y=y+10; y<40*v/2 ?  3
                  1        2

of
   1:  S1;
   2:  S2;
   3:  S3
end
```

This is identical to the more conventional program:

```
v = g(x,y);
x = f(x,y);
if x < v then
        begin  y:  =  y+10;
               if y< 40*v/2  then
                             S1;
                             else
                             S2;
        end;
        else
        S3;
```

This example illustrates the fact that any "computed case" statement has a corresponding representation using nested "if-then-else" statements and vice-versa.

As a program design and initial coding tool such a statement can enhance program readability by the elimination of complexly nested "if-then-else" statements. The various paths through a multi-level binary decision process are more naturally displayed by a two-dimensional tree structure than by even the most properly indented nested "if-then-else" sequence. An additional statement advantage of the computed case. statement occurs if several paths lead to the same action statements. Since in the nested "if-then-else" construct one must either repeat the action statements or reference them as a procedure. The latter solution is not always desirable since it requires additional computer time or memory space.

In their paper Franta and Maly also present an axiomatic definition of the computed case statement and discuss a number of implementation considerations (F4). Although Knuth (K3) and Dijkstra (D2) have called for the invention of such new program control constructs, our main purpose here is not to suggest additions to standard languages but rather to see how to use them to write more structured, more readable and therefore easier maintained programs.

4.2 A CASE FLOW ANALYSIS OF BVEQJ

Procedure BVEQJ of BVDDP receives control when a queued entry by the BVP I/O handler has been honored by the executive scheduler. This entry could be caused by a sweep report interrupt, a status report interrupt (radar charge, radar/range, and master clear) or a full buffer of video data (sweep reports and bracket detections). BVEQJ examines all I/O request completion codes for non-completion or time-out conditions. If all requests are completed BVEQJ checks to see if video processing is locked out. If so, the interrupt processor BVEQIN is called and control is returned to the executive scheduler. If video processing is not locked out, BVEX is called to do video stores and intermediate stores control and processing and BVEQIN is called to process SDC interrupts (sweep and status). BVEQJ next checks to see if the intermediate stores load is greater than light or if the signal data connector has filled a video stores buffer. If no, BV schedules itself on the equipment entrance

STATEMENT NO.	LABEL	STATEMENT FORM
1		if ( ) then goto L6
2		if ( ) then goto L10
3		if ( ) then goto L14
4		if ( ) then goto L18
5		goto L26
6	L6:	if ( ) then goto L25
7		if ( ) then goto L22
8		get ( ) to 0
9		goto L26
10	L10:	if ( ) then goto L25
11		if ( ) then goto L22
12		set ( ) to 0
13		goto L26
14	L14:	if ( ) then goto L25
15		if ( ) then goto L22
16		set ( ) to 0
17		goto L25
18	L18:	if ( ) then goto L25
19		if ( ) then goto L22
20		set ( ) to 0
21		goto L26
22	L22:	set ( ) to ( )
23		set ( ) to 1
24		call BVIMIC
25	L25:	call EXEC
26	L26:	if ( ) then goto L39
27		vary ( ) from 0 thru ( )
28		if ( ) then goto L30
29		goto L36
30	L30:	if ( ) then set ( ) to 0
31		then goto L36
32		set ( ) to ( )
33		set ( ) to 1
34		call BVIMIC
35		call EXEC

```
36          L36:    call BVEX
37                  call BVEQIN
38                  set ( ) to ( )
39          L39:    if ( ) then set ( ) to ( ) + 1
40                  if ( ) then goto L43
41                  if ( ) then goto L43
42                  goto L44
43          L43:    call EXEC
44          L44:    call EXEC
45                  return
```

FIGURE 4.1 THE STRUCTURE OF EVEQJ
IN CMS-2 LIKE NOTATION

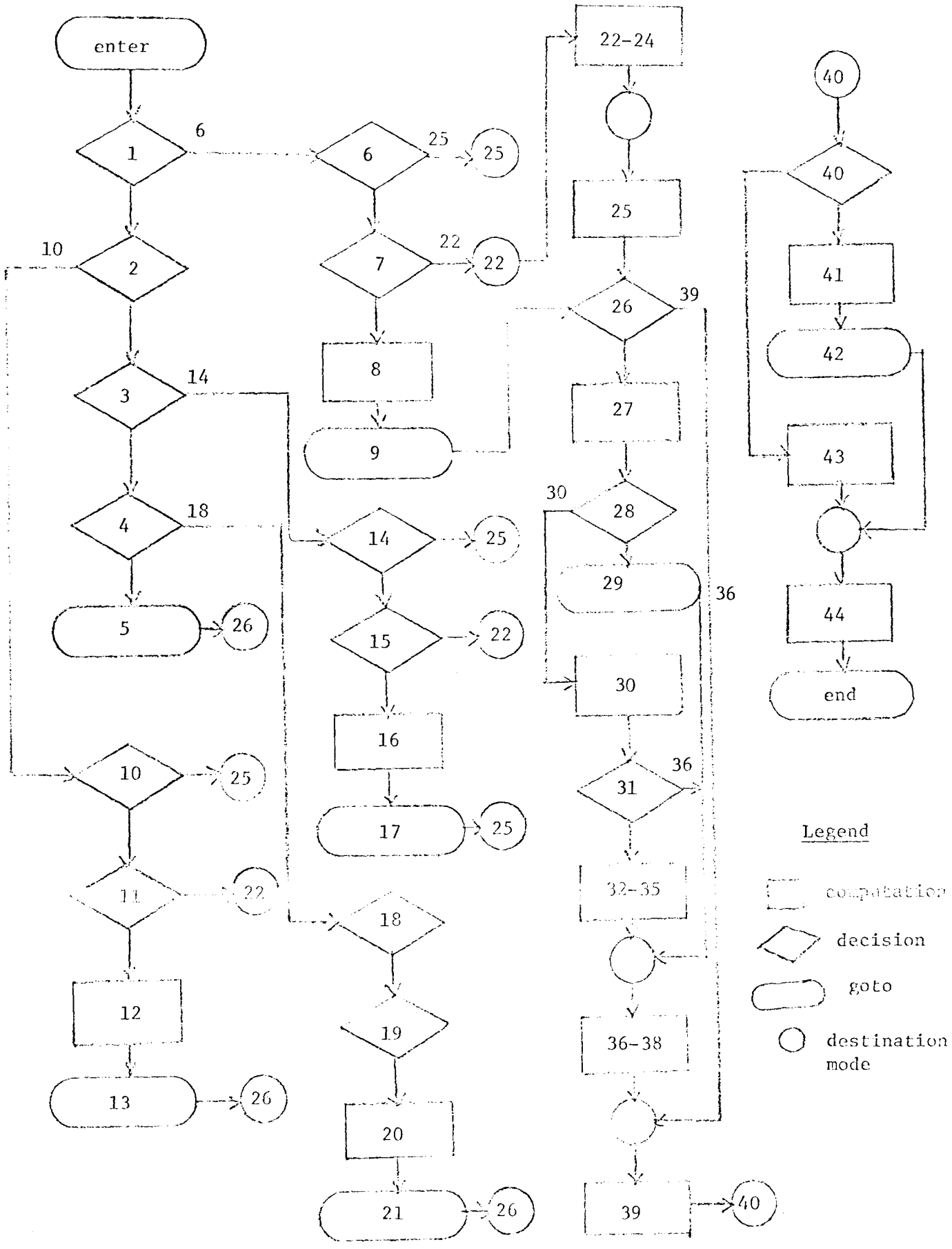


FIGURE 4.2 FLOWCHART OF BVEQJ BY STATEMENT NUMBERS OF FIGURE 4.1

queue. Control is returned to the executive scheduler.

The structure of BVEQJ is given in a CMS-2 like notation in Figure 4.1. The actual statements of the program are not germaine to an analysis of structure and are anyway classified confidential. A serious philosophical question does arise, however, when one reads the program description above and examines the program's structure in Figure 4.1. This is, what sense does it make to discuss the structure of a small module in isolation from its neighbors or from the system as a whole? Such an approach is clearly not "top-down", but nonetheless a general idea of a structuring technique, its effectiveness and the cost of the result in memory space and computer time can be gleaned from even a microscopic analysis of a typical module. It would, of course, guarantee a more significant result if one were to take an entire combat data system or even a major subsystem for analysis.

While BVEQJ was chosen more or less at random from BVDPP it was noted for its high density of goto statements. Even so, this module is not untypical of tactical code, particularly that involving program control, table control sensor or other I/O control functions. The flow chart of BVEQJ given in Figure 4.2 shows how really complex this small module is from a structural (rather than a computational) point-of-view. It would take a dedicated, patient programmer to understand this module and a courageous one to modify it. The analysis reported here began by structuring the lines numbers 1 through 44 of Figure 1 into a tree with branches at decision nodes. The analysis proceeds much like the node splitting technique discussed earlier but since the case flow decision trees contains references to code in the list below it is not necessary to duplicate so much code.

Figure 4.3 presents the form of procedure BVEQJ using the case flow statement. Only five statements are needed to represent the procedure using this new structured statement. Since one cannot compile the program of Figure 4.3 with any current high level language compiler the program is of value only to help one understand the procedure itself. This is done as a glance at the flow chart of Figure 4.4 will reveal. This flow chart also illustrates the code that would be repeated and cost memory space (if incline coded) or computer time (if reached via procedure call) if a node splitting technique were employed.

```
I.  caseflow                        1?

                          2?                      6?

                10?             3?          7?     L1

           11?        L1    4?     14?    L2    L3

      L13      L4         L5    18?    15?  L1

                             19?    L1 L6   L3

                          L3     L7


     of

     L1:    25;
     L2:     8;
     L3:    22; 23; 24; 25;
     L4:    12;
     L5:      ;
     L6:    16; 25;
     L7:    20
     end;


II.  caseflow                 26?

                          27; 28?       L1

                      30; 31;      L2

                  L3          L2


     of

     L1:      ;
     L2:    36; 37; 38;
     L3:    32; 33; 34; 35; 36; 37; 38;
     end;


III. 39;


IV.  if 40 then 41; 42 else 43;

 V.  44;
```

FIGURE 4.3 BVEQJ WRITTEN IN MULTILEVEL CASE STATEMENT NOTATION

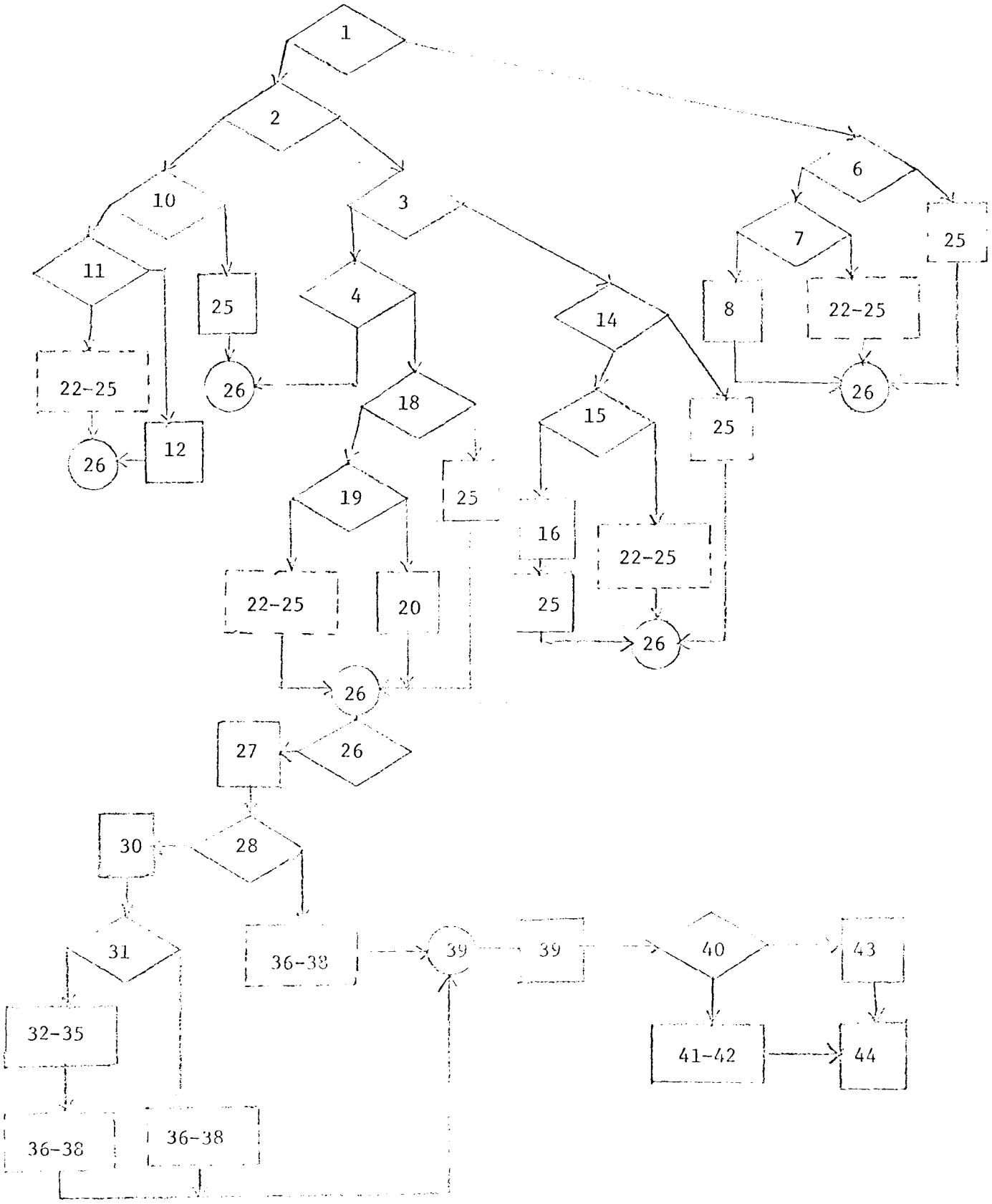


FIGURE 4.4 FLOW DIAGRAM OF THE BYEQ VERSION IN FIGURE 4.3. REPEATED CODE OUTLINED IN DASHED LINE BLOCKS.

```
if 1 then
     if 2 then
          if 10 then
               if 11 then 22, 23, 24, 25 else 12
               else 25
          else
          if 3 then
               if 14 then
                    begin
                    if 18 then
                         if 19 then 22, 23, 24, 25 else 20
                         else 25
                    end
               else begin
                    if 14 then
                         if 15 then 16; else 22, 23, 24
                         25;
                         end
          else
           if 6 then
               if 7 then 8 else 22, 23, 24, 25
               else 25
           if 26 then begin 27
                     if 28 then 30
                          else
                          if 31 then
                               begin  32, 33, 34, 35, 36, 37, 38, end
                               else
                               begin 36, 37, 38 end
                     end
           39
           if 40 then 41, 42 else 43
           44
```

FIGURE 4.5 IMPROVED VERSION OF BVEQJ IN INDENTED ALGOL-LIKE NOTATION

A more conventional yet still structured form of BVEQJ appears in Figure 4.5 using an ALGOL-like block structured language with indentation. This program was derived in a straightforward fashion from the flow chart in Figure 4.4.

4.3 COMPARISONS AND COST ANALYSIS

Both structured versions of BVEQJ generate more code than the original program when hand compiled into AN/UYK-7 code. Figure 4.6 presents the cost of repeated code in additional instructions required. The original version compiled to 85 machine language instructions so even the extremely efficient case flow technique has a cost of nearly 10% repeated code; a more conventional approach costs nearly 30% in added code. The time overhead for procedure calls in present day high level languages is so high that the time penalty for taking such small bits of repeated code out-of-line would be even more prohibitive than the space penalty seen in Figure 4.6.

STRUCTURAL MODE	INCREASED COSTS IN STATEMENTS REPEATED			INCREASED COST IN MACHINE LANGUAGE
	No.	Copies	Words/Copy	
Original form unstructured, unrestrained use of goto's	-	-	-	-
Case Flow	25	1	1	3
Structure	36	1	1	1
	37	1	1	1
	38	1	3	3
				8 TOTAL
Indented	22	4	2	6
ALCOL-like	23	4	2	6
Structure	24	4	1	3
	25	6	1	5
	36	2	1	1
	37	2	1	1
	38	2	3	3
				25 TOTAL

FIGURE 4.6 STRUCTURED PROGRAM COSTS FOR BVEQJ

5. CONCLUSION

The conclusion of this study, which was a lengthy review of the literature that has developed in structured programming since 1964 plus some experiments with reprogramming program fragments written in several different programming languages into structured form, is that structured programming is here to stay but that it's still much to early in the development of the subject to commit to absolute rules.

Much of the early literature on the subject centered around elimination of the goto statement since free use of this statement complicates program topology and tends to make programs unreadable. A number of clever new program constructs have been suggested and a few new goto-less languages even designed and implemented but the need for something like a goto still persists. Most of these alternatives have a goto equivalent which turns out to be semantically the same thing as the goto but with restricted scope. The reason for introducing such a mechanism is clearly efficiency since disallowing goto's in inner loops and other time critical areas of code causes either code duplication and waste of memory space, the use of inefficient Boolean variables, or calls to small procedures and thus waste of procedure time.

The authors of this report think that the current activity and research in structured programming will lead to the development of programming languages which allow both the convenient coding of top-down and structured program designs and programmer directed local optimization of critical code. Some more experimentation with basic structures will be needed before an optimum set is settled on (but the goto will probably be retained in these languages for coding new fundamental structures as macros). Current techniques for automatic optimization will probably be replaced by manual methods whereby the programmer directs the optimizing pass of the compiler via source language declaration. (K3) This will be a great boon to the realtime programmer since he can have the best features of both program manageability and program efficiency. He should be able to direct the compiler to produce even the equivalent of careful hand coding in his critical code sequences.

Finally, the authors of this study recommend to the tactical coder in today's languages that he design top-down and structured, that he code top-down and structured, that he avoid goto statements whenever possible, but that if he must use them for efficiency that he restrict their scope to be local and if this technique doesn't allow him to meet requirements that he code time critical sections by hand as he has always done.

It should be noted that although this report has addressed structured programming generally, its specific emphasis has been on the goto issue and thus it has only faced a part of the overall problem. Many of the major techniques of structured programming have been applied in the development of tactical data processing software. The efficiency issue has always been a key one for tactical code thus the recitation of a goto-less formula by the highpriests of structured programming was seen as over emphasis by the tactical programmer. Thus, we have primarily addressed this concern although it is only one aspect of the overall subject.

6. AN ANNOTATED BIBLIOGRAPHY ON STRUCTURED PROGRAMMING

In recent years the literature on the seemingly discursive and controversial subject of structured programming has grown at an exponential rate. And yet in the face of this abundance of information, to the minds of many a definitive assessment of the subject has so far eluded clear statement. In a real sense, the current "definition" is given by the totality of the literature on the subject. In an effort to bring this totality of information to a larger audience, this chapter presents an annotated bibliography of papers and articles on the subject.

The full texts of all items listed are readily available in periodicals or enjoy widespread private circulation. An attempt has been made to provide a balance between items of theoretical and practical interest. In both categories listings of seminal as well as "bandwagon" works have been included. No attempt has been made to label either as both types can contribute to one's understanding of the subject.

The listing cannot be assumed to be complete, although absence of a truly significant work can be assumed to imply that the authors are ignorant of its existence. In any case, the listing is sufficiently voluminous to convey the consensus, intents and direction of the subject of structured programming.

The bibliography is concluded by a listing of several items without abstract or comment. The latter are either books, the summaries of documents which require more than a single paragraph of annotation or items unavailable for inspection.

Finally, the authors apologize now to any authors whose works were omitted, as that omission is not intended as a negative comment on their work.

A1 E. Ashcroft and Z. Manna, "The translation of 'go to' programs to 'while' programs," Proc. IFIP Congress 71, Ljubljana, August 1971.

In this paper it is shown that every flowchart program can be written without go to statements by using while statements. The main idea is to introduce new variables to preserve the values of certain variables at particular points in the program; or alternatively, to introduce special boolean variables to keep information about the course of the computation. The 'while' programs produced yield the same final results as the original flowchart program but need not perform computations in exactly the same way. However, the new programs do preserve the 'topology' of the original flowchart program, and are of the same order of efficiency. The translation cannot be done in general without using auxiliary variables.

B1 F. T. Baker and H. D. Mills, "Chief programmer teams," Datamation, V. 19, N. 12, December 1973.

This fundamental change in the managerial framework of production programming structures programming work into specialized jobs, defines relationships among specialists, and stresses discipline and teamwork.

B2 F. T. Baker, "Chief programmer teams," IBM Syst. Jour., V. 11, N.1, 1972.

IBM has developed an idea which goes in approach from a loosely structured "soccer team" of programmers to a highly structured "surgical team" of several technical and clerical specialists who employ strict operational procedures. A Team nucleus, consisting of a Chief Programmer, Backup Programmer and a Programming Secretary, specifies and supervises all programming operations in complete detail. Initial experience indicates that personnel in such Chief Programmer Teams can be twice as productive as in present programming groups. But, even more importantly in many situations, the reliability and maintainability of the programs produced is unprecedented. The technical procedures are based on new mathematical foundations of Structured Programming, which provide for a new level of precision rigor in program design, construction, and validation. The clerical procedures change programming from "bench work" to "assembly line" operations, using a Programming Production Library, which holds a developing system in a central, visible form, where architects, programmers, analysts, technicians, secretaries, and others can bring their special skills to bear on a common project.

B3 V. R. Basili, A. J. Turner, "Experiences with a simple structured programming language," Proc. Fourth Symposium on Computer Science Education, February 1974.

This paper is concerned with some experiences obtained in the use of a structured programming language in the computer science curriculum at the University of Maryland. The language used was SIMPL-X, a language designed and implemented at the University of Maryland. SIMPL-X was designed to be a transportable, extendable, compiler-writing language that was to be the base language for a family of programming languages. However, some of the design criteria for SIMPL-X have made it a reasonable language for use in programming courses at all levels. These criteria include the requirements that the language

1) have a "simple" control structure and require only a "simple" run time environment.

2) conform to the standards of structured programming and modular program design.

3) support and encourage the writing of readable, well-commented programs.

4) be translatable into efficient object code for most machines.

This paper summarizes the SIMPL-X language and some of the experiences resulting from its use at the University of Maryland.

B4 G. V. Bochmann, "Multiple exits from a loop without the GO TO," CACM, V. 16, N. 7, July 1973.

It has been pointed out that "goto" free programs tend to be easier to understand, allow better optimization by the compiler, and are better suited for an eventual proof of correctness. On the other hand, the "goto" statement is a flexible tool for many programmers. Most programming languages have constructs which allow the programmer to write control flows that occur frequently without the use of a "goto." In particular, the language Pascal [3] contains, besides the "goto," the following control structures: "if-then-else, case, while-do, repeat-until," stepping loop.

B5 C. Bohm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," CACM, V. 9, N. 5, May 1966.

In the first part of the paper, flow diagrams are introduced to represent mappings of a set into itself. Although not every diagram is decomposable into a finite number of given base diagrams, this becomes true at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two. In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing machine provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by

composition and iteration from the two machines λ and "R". That family is a proper subfamily of the whole family of Turing machines.

Comment: Much referenced work. Pointed to as work of much practical value. Closer inspection suggests otherwise. See remarks in K3.

B6 R. M. Burstall, "Proving properties of programs by structural induction,"

Computer Jour., V. 12, N. 1, February 1969.

This paper discusses the technique of structural induction for proving theorems about programs. This technique is closely related to recursion induction but makes use of the inductive definition of the data structures handled by the programs. It treats programs with recursion but without assignments or jumps. Some syntactic extensions to Landin's functional programming language ISWIM are suggested which make it easier to program the manipulation of data structures and to develop proofs about such programs. Two sample proofs are given to demonstrate the technique, one for a tree sorting algorithm and one for a simple compiler for expressions.

C1 R. N. Chanon, "On a measure of program structure," Proc. Colloque Sur la Programmation, Paris, April 1974.

Not every piece of software which consists of a collection of small programs has good structure. Nor do informal methods necessarily guarantee good structure. The goal of this thesis has been to investigate the behavior of a mathematical tool - entropy loading - as a measure of the goodness of structure and as a guide which can help to preserve good structure in a collection of programs which constitutes the decomposition of a piece of software.

C2 R. Lawrence Clark, "A linguistic contribution to GOTO-less programming," Datamation, V. 19, N. 12, December 1973.

We don't know where to GOTO if we don't know where we've COME FROM. This linguistic innovation lives up to all expectations.

Comment: Tongue in cheek. Very amusing.

C3 M. Clint and C. A. R. Hoare, "Program proving: jumps and functions," Acta Informatica, V. 1, 1972, pp. 214-224.

Proof methods adequate for a wide range of computer programs have been expounded. This paper develops a method suitable for programs containing functions, and a certain kind of jump. The method is illustrated by the proof of a useful and efficient program for table lookup by logarithmic search.

C4 M. Clint, "Program proving: coroutines," Acta Informatica, V. 2, 1973, pp. 50-63.

Proof methods adequate for a wide range of computer programs have been given. This paper develops a method suitable for programs which incorporate coroutines. The implementation of coroutines described follows closely that given in SIMULA, a language in which such features may be used to great advantage. Proof rules for establishing the correctness of coroutines are given and the method is illustrated by the proof of a useful program for histogram compilation.

Comment: Especially valuable to those concerned with simulation and operating systems.

C5 P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with 'Readers' and 'Writers'," CACM, V. 14, N. 10, October 1971.

The problem of the mutual exclusion of several independent processes from simultaneous access to a "critical section" is discussed for the case where there are two distinct classes of processes known as "readers" and "writers." The "readers" may share the section with each other, but the "writers" must have exclusive access. Two solutions are presented: one for the case where we wish minimum delay for the readers; the other for the case where we wish writing to take place as early as possible.

Comment: Nicely done. Good demonstration of code for cooperating processes.

D1 P. J. Denning, "Is it not time to define 'structured programming'?," Operating Systems Review, V. 8, N. 1, January 1974, pp. 6-7.

Comment: Points out absence of definition and that we (programmers) have been protagonists and antagonists of something undefined. Capsulizes common impressions of what is.

D2 E. W. Dijkstra, "Notes on structured programming," EWD 249, Technical University, Eindhoven, Netherlands, 1969. Also published in Structured Programming, Academic Press, London, 1972.

A rambling, rather philosophical discussion of the issues raised by the technique of structured programming. The author's points are often well illustrated by analogies as well as programming examples.

D3 Edsger W. Dijkstra, "A simple axiomatic basis for programming language constructs," EWD 372, Unpublished.

The semantics of a program can be defined in terms of a predicate transformer associating with any post-condition (characterizing a set of final states) the corresponding weakest pre-condition (characterizing a set of initial states). The semantics of a programming language can be defined by regarding a program text as a prescription for constructing its corresponding predicate transformer. Its conceptual simplicity, the modest amount of mathematics needed and its constructive nature seem to be its outstanding virtues. In comparison with alternative approaches it should be remarked, firstly, that all nonterminating computations are regarded as equivalent and, secondly, that a program construct like the goto-statement falls outside its scope; the latter characteristic, however, does not strike the author as a shortcoming, on the contrary, it confirms him in one of his prejudices.

Comment: Very interesting notions but quite avant-garde.

D4 Edsger W. Dijkstra, "Guarded commands, non-determinacy and a calculus for the derivation of programs," Unpublished.

So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow non-deterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be shown.

Comment: Must reading. In a real sense a follow-up to D3.

D5 E. W. Dijkstra, "A Constructive approach to the problem of program correctness," BIT 8, 1968, pp. 174-186.

As an alternative to methods by which the correctness of given programs can be established a posteriori, this paper proposes to control the process of program generation such as to produce a priori correct programs. An example is treated to show the form that such a control might then take. This example comes from the field of parallel programming; the way in which it is treated is representative of the way in which a whole multiprogramming system has actually been constructed.

D6 Edsger W. Dijkstra, "The humble programmer," CACM, V. 15, N. 10, Oct. 1972.

A study of program structure has revealed that programs--even alternative programs for the same task and with the same mathematical content--can differ tremendously in their intellectual manageability. A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program. These rules are discussed in this lecture transcript.

Comment: Must reading.

D7 E. W. Dijkstra, "Recursive Programming," Programming Systems and Languages, (Ed. Rosen, S.), McGraw-Hill, New York, 1967.

If every subroutine has its own private fixed working space, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need "simultaneously," and the available memory space is therefore used rather uneconomically. Furthermore--and this is a more serious objection--it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on. The author describes the principles of a program structure for which these two objections no longer hold. In the first place he sought a means of removing the second restriction, for this essentially restricts the admissible structure of the program; hence the name "Recursive Programming." More efficient use of the memory as regards the internal working spaces of subroutines is a secondary consequence not without significance. The solution can be applied under perfectly general conditions, e.g., in the structure of an object program to be delivered by an ALGOL 60 compiler. The fact that the proposed methods tend to be rather time-consuming on an average present day computer, may give a hint in which direction future design might go.

D8 E. W. Dijkstra, "Solution of a problem in concurrent programming control," CACM, V. 8, N. 9, September 1965.

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Comment: Perhaps first published paper on synchronization of processes.

D9 James R. Donaldson, "Structured programming," Datamation, V. 19, N. 12, December 1973.

The fundamental message is "simplify your control paths."

Comment: Good reading for general information.

D10 O. J. Dahl and C. A. R. Hoare, "Hierarchical program structures," Structured Programming, Academic Press, London, 1972.

This monograph explores certain ways of program structuring and points out their relationship to concept modelling. Use is made of the programming language SIMULA 67 with particular emphasis on structuring mechanisms. SIMULA 67 is based on ALGOL 60 and contains a slightly restricted and modified version of ALGOL 60 as a subset. Additional language features are motivated and explained informally when introduced.

D11 P. J. Denning, "Is 'Structured Programming' Any Longer the Right Term?" Operating Systems Review 8, 4, October 1974, pp. 4-6.

This letter is a follow-up to Denning's earlier letter (see D1); he now considers a definitive treatment of structured programming to be unachievable. That is, there is no fixed set of rules by which clear, understandable, provable programs can be constructed.

D12 E. W. Dijkstra, "Goto statement considered harmful," CACM, pp. 147-148, 538, 541, March 1968.

F1 D. A. Fisher, "A survey of control structures in programming languages," Sigplan Notices, V. 7, N. 11, November 1972.

The control structure of programming languages and their development are examined. Languages studied range from machine and assembly languages to procedure and problem-oriented languages. The emphasis, however, is on the control structures themselves, whether in current languages or proposed. Both implicit global interpretation rules for programming languages and explicit control operations are discussed. Many control structures developed through specialization from a small set of primitive sequential control operations. Specific control structures and mechanisms examined include activities, broadcast control, conditionals, constraint expressions, co-routines, critical sections, distributive operators, dynamic instruction modification, expressions, generators, implicit co-routines, implicit sequencing, iterative control, indivisibility, interleaved execution, the goto, macros, multipass algorithms, multiple sequential control, mutual exclusion, mutual subroutines, nonbusy waiting, nondeterministic control, open subroutines, parallel assignments, parallel processing, procedures, pseudo-parallel control, recursion, reentrant code, relative continuity, semaphores, sequential controls, shared procedures, simultaneous assignments, statements, subroutines, synchronization, syntax macros, time-sharing and backtracking.

F2 Clinton R. Foulk, "Yet another attempt to define 'structured programming,'" Operating System Review, V. 8, N. 3, July 1974.

Comment: Another response to the Denning Letter to Operating Systems Review.

F3 W. R. Franta and Kurt Maly, "A Multilevel Flow Control Statement," Technical Report 74-20, University of Minnesota Computer Science Department, September 1974.

The premise of this report is that the nested 'if-then-else' construct occurs frequently and that these constructs if nested deeply are detremental to program readability. To mitigate this situation the authors propose and describe a control structure which provides for the description of a multilevel binary decision process via a binary tree. Examples are presented to promote their viewpoint, and a formal inference rule is supplied.

G1 Philip Gilbert and W. J. Chandler, "Interference between communicating parallel processes," CACM, V. 15, N. 6, June 1972.

Various kinds of interference between communicating parallel processes have been examined by Dijkstra, Knuth, and others. Solutions have been given for the mutual exclusion problem and associated subproblems, in the form of parallel programs, and informal proofs of correctness have been given for these solutions. In this paper a system of parallel processes is regarded as a machine which proceeds from one "state S" (i.e., a collection of pertinent data values and process configurations) to a next state "S'" in accordance with a "transition rule S→S'." A set of such rules yields sequences of states, which dictate the system's behavior. The mutual exclusion problem and the associated subproblems are formulated as questions of inclusion between sets of states, or of the existence of certain sequences. A mechanical proof procedure is shown, which will either verify or discredit an attempted solution, with respect to any of the interference properties. It is shown how to calculate transition rules from the "partial rules" by which the individual processes operate. The formation of partial rules and the calculation of transition rules are both applicable to hardware processes as well as to software processes, and symmetry between processes is not required.

G2 David Gries, "What should we teach in an introductory programming course?," Proc. 4th Symposium on Computer Science Education, February 1974.

An introductory course (and its successor) in programming should be concerned with three aspects of programming: (1) How to solve problems, (2) How to describe an algorithmic solution to a problem, (3) How to verify that an algorithm is correct. The author discusses mainly the first two aspects. The third is just as important, but if the first two are carried out in a systematic fashion, the third is much easier than commonly supposed.

G3 David Gries, "On Structured Programming - A Reply to Smoliar," ACM Forum, CACM, V. 17, N. 11, November 1974, pp. 655-657.

This letter is a response to Smoliar's letter (see S5) and an attempt to give a general definition of, and a cross-section of research problems in, structured programming.

H1 A. N. Habermann, "Critical comments on the programming language Pascal," Acta Informatica, Vol. 3, 1973, pp. 47-57.

The programming language Pascal is claimed to be more suitable than other languages for "teaching programming as a systematic discipline." However, an investigation of the Reports on the Pascal language reveals that it suffers as much from ill-defined constructs as many of the languages to which it is supposed to offer an alternative. Problems with the language are caused primarily by the confusion of ranges, types and structures and by the phenomena associated with goto statements.

Comment: Compare this report with Wirth's, W4.

H2 A. N. Habermann, "Synchronization of communicating processes," CACM, V. 15, N. 3, March 1972.

Formalization of a well-defined synchronization mechanism can be used to prove that concurrently running processes of a system communicate correctly. This is demonstrated for a system consisting of many sending processes which deposit messages in a buffer and many receiving processes which remove messages from that buffer. The formal description of the synchronization mechanism makes it very easy to prove that the buffer will neither overflow nor underflow, that senders and receivers will never operate on the same message frame in the buffer nor will they run into a deadlock.

H3 P. Brinch Hansen, "Structured multiprogramming," CACM, V. 15, N. 7, July 1972.

This paper presents a proposal for structured representation of multiprogramming in a high level language. The notation used explicitly associates a data structure shared by concurrent processes with operations defined on it. This clarifies the meaning of programs and permits a large class of time-dependent errors to be caught at compile time. A combination of critical regions and event variables enables the programmer to control scheduling of resources among competing processes to any degree desired. These concepts are sufficiently safe to use not only within operating systems but also within user programs.

Comment: Must reading.

H4 P. Henderson and R. Snowdon, "An experiment in structured programming," BIT 12, 1972, pp. 38-53.

The construction of a program to solve a simple problem, written using a top-down structural approach, is described. An independent analysis of this program is provided commenting on the possible problems that arise from the use of such a technique.

Comment: Discusses an error found in a structured program.

H5 P. Henderson, and P. Quarendon, "Finite state testing of structured programming," Proc. Colloque Sur La Programmation, Paris, April 1974, pp. 56-59.

Comment: Discusses the simulation needs when testing incomplete programs which are being developed in a top-down manner.

H6 C. A. R. Hoare, "A note on the for statement," BIT V. 12, N. 3, 1972, pp. 334-341.

This note discusses methods of defining the for statement in high level languages and suggests a proof rule intended to reflect the proper role of a for statement in computer programming. It concludes with a suggestion for possible generalization.

H7 C. A. R. Hoare, "The quality of software," Software - Practice and Experience, V. 2, 1972, pp. 103-105.

The main problem in the design of any engineering product is the reconciliation of a large number of strongly competing objectives. In the case of general purpose computer software, he has made a list of no less than seventeen:

1) Clear definition of purpose
2) Simplicity of use
3) Ruggedness
4) Early availability
5) Reliability
6) Extensibility and improvability in light of experience
7) Adaptability and easy extension to different configurations
8) Suitability to each individual configuration of the range

9) Brevity
10) Efficiency (speed)
11) Operating ease
12) Adaptability to wide range of applications
13) Coherence and consistency with other programs
14) Minimum cost to develop
15) Conformity to national and international standards
16) Early and valid sales documentation
17) Clear, accurate and precise user's documents

H8 C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," Acta Informatica, V. 2, 1973, pp. 335-355.

The axiomatic definition method proposed in reference H12 is extended and applied to define the meaning of the programming language PASCAL W4. The whole language is covered with the exception of real arithmetic and go to statements.

H9 C. A. R. Hoare, "Hints on programming language design," Stanford Artificial Intelligence Laboratory, Computer Science Department Report No. CS-403, Stanford University, October 1973.

This paper presents the view that a programming language is a tool which should assist the programmer in the most difficult aspects of his art, namely program design, documentation, and debugging. It discusses the objective criteria for evaluating a language design, and illustrates them by application to language features of both high level languages and machine code programming. It concludes with an annotated reading list, recommended for all intending language designers.

Comment: Good exposition of widely known points.

H10 C. A. R. Hoare, "Proof of a structured program: the sieve of Eratosthenes," Computer Jour., V. 15, N. 4, 1972, pp. 321-325.

This paper illustrates a method of constructing a program together with its proof. By structuring the program at two levels of abstraction, the proof of the more abstract algorithm may be completely separated from the proof of the concrete representation. In this way, the overall complexity of the proof is kept within more reasonable bounds.

H11 C. A. R. Hoare, "Proof of a program: FIND," CACM, V. 14, N. 1, January 1971, pp. 39-45.

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A

systematic technique is described for constructing the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

Comment: The purpose of the program Find [4] is to find that element of an array A[1:N] whose value is fth in order of magnitude; and to rearrange the array in such a way that this element is placed in A[f]; and furthermore, all elements with subscripts lower than f have lesser values, and all elements with subscripts greater than f have greater values. Thus on completion of the program, the following relationship will hold:

$$A[1],A[2],\ldots,A[f-1]\leq A[f]\leq A[f+1],\ldots,A[N]$$

This relation is abbreviated as Found.

H12 C. A. R. Hoare, "Proof of correctness of data representations," Acta Informatica, V. 1, 1972, pp. 271-281.

In the development of programs by stepwise refinement, the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation. This paper suggests an automatic method of accomplishing the transition between an abstract and a concrete program, and also a method of proving its correctness; that is, of proving that the concrete representation exhibits all the properties expected of it by the "abstract" program. A similar suggestion was made more formally in algebraic terms; however, a more restricted definition may prove to be more useful in practical program proofs. If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. Since abstract programs are usually very much shorter and easier to prove correct, the total task of proof has been considerably lightened by factorising it in this way. Furthermore, the two parts of the proof correspond to the successive stages in program development, thereby contributing to a constructive approach to the correctness of programs.

H13 C. A. R. Hoare, An axiomatic basis for computer programming," CACM, V. 12, N. 10, October 1969.

> In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.
>
> Comments: Must reading, often referenced.

H14 C. A. R. Hoare, "Notes on data structuring," Structured Programming, Academic Press, London, 1972.

> The second section explains the concept of type, which is essential to the theory of data structuring; and relates it to the operations and representations which are relevant to the practice of computer programming. Subsequent sections deal with particular methods of structuring data, progressing from the simpler to the more elaborate structures. Each structure is explained informally with the aid of examples. Then the manipulation of the structure is defined by specifying the set of basic operations which may be validly applied to the structure. Finally, a range of possible computer representations is given, together with the criteria which should influence the selection of a suitable representation on each occasion. The last section puts the whole exposition on a rigorous theoretical basis by formulating the axioms which express the basic properties of data structures.

H15 M. Hopkins, "A case for the goto," Proceedings ACM '72, Boston, August 1972.

> In recent years there has been much controversy over the use of the goto statement. This paper, while acknowledging that goto has been used too often, presents the case for its retention in current and future programming languages.

K1 B. W. Kernighan and P. J. Plauger, "Programming Style," Proc. 4th Symposium on Computer Science Education, February 1974.

> Programs written with good style are easier to read and understand, and typically smaller and more efficient than those written badly, regardless of the language used. Yet most programmers have never been taught programming style --as proof we need only look at their programs. In this paper we will discuss several principles of programming style, illustrating these points by criticizing and rewriting some real programs. The examples are all taken verbatim from programming textbooks, and the revisions have all been tested.

K2 D. E. Knuth and R. W. Floyd, "Notes on avoiding 'go to' statements," Information Processing Letters 1, North-Holland, Amsterdam, 1971. pp. 23-31.

> During the last decade there has been a growing sentiment that the use of "go to" statements is undesirable, or actually harmful. This attitude is apparently inspired by the idea that programs expressed solely in terms of conventional iterative constructions ("for," "while," etc.) are more readable and more easily proved correct. In this note a few exploratory observations are made about the use and disuse of go to statements, based on two typical programming examples (from "symbol table searching" and "backtracking").

K3 Donald E. Knuth, "Structured programming with go to statements," Unpublished as of November, 1974. To be in December 1974 Computing Reviews.

> A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b) a methodology of program design, beginning with readable and correct but possibly inefficient programs that are systematically transformed if necessary into efficient and correct but possibly less readable code. The discussion brings out opposing points of view about whether or not go to statements should be abolished; some merit is found on both sides of this question. Finally an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.
>
> Comment: Must reading. In an attempt to retain considerations of efficiency in programs, the complete elimination of go to's is reassessed.

K4 Donald E. Knuth, "A review of structured programming," STAN-CS-73-371, June 1973.

Comment: An assessment and review of the book Structured Programming [D2]. One section of the report deals with each of the three sections of the book. Must reading.

K5 S. Rao Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets," Proc. Fourth Symposium on Operating System Principles in Operating Systems Review, V. 7, N. 4, October 1973.

Recently various attempts have been made to study the limitations of Dijkstra's Semaphore Primitives for the synchronization problem of cooperating sequential processes. Patil proves that the semaphores with the P and V primitives are not sufficiently powerful. He suggests a generalization of the P primitive. It is proved that certain synchronization problems cannot be realized with the above generalization and even with arrays of semaphores. It is also shown that even the general Petri nets will not be able to handle some synchronization problems, contradicting a conjecture of Patil.

K6 M. M. Kessler, "Implementation of macros to permit structured programming in OS/360," IBM CONCEPT Report 14, Federal Systems Division, Gaithersburg, MD, December 1970.

H. D. Mills proposed that the concept of block-structured programming be introduced into assembly language programming by producing a set of structure macros. In addition, he proposed that these macros be implemented as simple and small entities rather than as large or complex ones, and in order to assist the developer of the macros in reaching these goals, he suggested the following certain key implementation rules. One of these is that all macros which are implemented must represent a proper flow chart. Inherent in the prefix "proper" is that each such flow chart defined by a related group of macros (macro set) has a single input and a single output. Another concept involved the introduction of unique terminators for each macro set instead of a universal END macro (or its equivalent) for all sets. Thus the macro set IF, ELSE, ENDIF has a unique ENDIF terminator to indicate the point at which all branches produced as a result of the execution of the previous members of the set join together.

K7 B. W. Kennington and P. J. Plauger, The Elements of Programming Style, (New York: McGraw-Hill Book Co., 1974) 147 pp.

This book is a study of a number of "actual" programs, each of which provides one or more lessons in style. The authors discuss the shortcomings of each example, rewrite it in a better way, then draw a general rule from the specific case. The approach is pragmatic and down-to-earth; it is oriented more towards improving current programming practice than in setting up an elaborate theory of how programming should be done. Consequently, this book can be used as a supplement in a programming course at any level, or as a refresher for experienced programmers. The examples are all in Fortran and PL/1, since these languages are widely used and are sufficiently similar that a reading knowledge of one means that the other can also be read well enough. The principles of style, however, are applicable in all languages, including assembly codes.

K8 D. E. Knuth, "Computer Programming as an Art," CACM V.17, No. 12, pp. 667-673, December 1974.

The meaning of the word art is examined, the relationship between art and science and art discussed and then the bulk of the paper relates the observations made to programming style. In this latter effort attention is focused on the need to provide beautiful tools for programming, and it is pointed out that beautiful can imply simple but properly conceived. As with much of Knuth's work the perspective the paper conveys is perceptive, elegent, concise, and above all useful.

L1 B. M. Leavenworth, "Programming with(out) the GOTO," Sigplan Notices, V. 7, N. 11, November 1972.

A brief history of the goto controversy (retention or deletion of the goto statement) is presented. After considering some of the theoretical and practical aspects of the problem, a summary of arguments both for and against the goto is given.

L2 B. H. Liskov, "Guidelines for the design and implementation of reliable software systems," ESD-TR-72-164, MTR-2345, MITRE Corp., February 1973, (AD-757905).

This document describes experimental guidelines governing the production of reliable software systems. Both programming and management guidelines are proposed. The programming guidelines are intended to enable programmers to cope with a complex system effectively. The management guidelines describe an organization of personnel intended to enhance the effect of the programming guidelines.

L3 B. H. Liskov, "A design methodology for reliable software systems," Proc. FJCC, 1972, pp. 191-199.

Any user of a computer system is aware that current systems are unreliable because of errors in their software components. While system designers and implementers recognize the need for reliable software, they have been unable to produce it. For example, operating systems such as OS/360 are released to the public with hundreds of errors still in them. A project is underway at the MITRE Corporation which is concerned with learning how to build reliable software systems. Because systems of any size can always be expected to be subject to changes in requirements, the project goal is to produce not only reliable software, but readable software which is relatively easy to modify and maintain. This paper describes a design methodology developed as part of that project.

Comment: Must reading.

L4 Barbara H. Liskov, "The design of the Venus Operating System," CACM, V. 15, N. 3, March 1972.

The Venus Operating System is an experimental multiprogramming system which supports five or six concurrent users on a small computer. The system was produced to

test the effect of machine architecture on complexity of software. The system is defined by a combination of microprograms and software. The microprogram defines a machine with some unusual architectural features; the software exploits these features to define the operating system as simply as possible. In this paper the development of the system is described, with particular emphasis on the principles which guided the design.

L5 Ralph L. London, "Proving programs correct: some techniques and examples," BIT, V. 10, N. 2, 1970, pp. 168-182.

Proving the correctness of computer programs is justified as both advantageous and feasible. The discipline of proof provides a systematic search for errors, and a completed proof gives sufficient reasons why the program must be correct. Feasibility is demonstrated by exhibiting proofs of five pieces of code. Each proof uses one or more of the illustrated proof techniques of case analysis, assertions, mathematical induction, standard prose proof, sectioning and a table of variable value changes. Proofs of other programs, some quite lengthy, are cited to support the claim that the techniques work on programs much larger than the examples of the paper. Hopefully, more programmers will be encouraged to prove programs correct.

L6 R. L. London, "Treesort 3: Proof of algorithms -- A new kind of certification," CACM, V. 13, N. 6, June 1970, pp. 371-373.

The certification of an algorithm can take the form of a proof that the algorithm is correct. As an illustrative but practical example, Algorithm 245, TREESORT 3 for sorting an array, is proved correct. Since suitable techniques now exist for proving the correctness of many algorithms, it is possible and appropriate to certify algorithms with a proof of correctness. This certification would be in addition to, or in many cases instead of, the usual certification. Certification by testing still is useful because it is easier and because it also provides, for example, timing data. Nevertheless the existence of a proof should be welcome additional certification of an algorithm. The proof shows that an algorithm is debugged by showing conclusively that no bugs exist.

M1 Edward F. Miller, Jr., and George E. Lindamood, "Structured programming: top-down approach," Datamation, V. 19, N. 12, December 1973.

Structured programming is a technique that reduces a program's complexity, increases its clarity, and results in easy maintenance.

M2 Harlen D. Mills, "Mathematical foundations for structured programming," IBM FSD Report FSC72-6012, Gaithersburg, MD, February 1972.

E. W. Dijkstra originated a set of ideas and a series of examples for clear thinking in the construction of programs. These ideas are powerful tools in mentally connecting the static text of a program with the dynamic process it invokes in execution. This new correspondence between program and process permits a new level of precision in programming. Indeed, it is contended here that the precision now possible in programming will change its industrial characteristics from a frustrating, trial and error activity to a systematic, quality controlled activity. However, in order to introduce and enforce such precision programming as an industrial activity, the ideas of structured programming must be formulated as technical standards, not simply as good ideas to be used when convenient, but as basic principles which are always valid. A good example of a technical standard occurs in logic circuit design. There, it is known, from basic theorems in boolean algebra, that any logic circuit, no matter how complex its requirement, can be constructed using only AND, OR, and NOT gates.

M3 Harlen Mills, "Top down programming in large systems," Debugging Techniques in Large Systems (Ed. Rustin, Randall), Prentice-Hall, Englewood Cliffs, NJ, 1971.

Structured programming can be used to develop a large system in an evolving tree structure of nested program modules, with no control branching between modules except for module calls defined in the tree structure. By limiting the size and complexity of modules, unit debugging can be done by systematic reading, and the modules executed directly in the evolving system in a top down testing process.

M4 H. D. Mills, "How to write correct programs and know it," IBM Report FSC 73-5008, Federal Systems Division, Gaithersburg, MD, February 1973.

There is no foolproof way to ever know that you have found the last error in a program. So the best way to acquire the confidence that a program has no errors is never to find the

first one, no matter how much it is tested and used. It is an old myth that programming must be an error-prone, cut-and-try process of frustration and anxiety. But there is a new reality that you can learn to consistently write programs which are error free in their debugging and subsequent use. This new reality is founded in the ideas of structured programming and program correctness, which not only provide a systematic approach to programming but also motivate a high degree of concentration and precision in the coding subprocess.

M5 R. C. Mc Henry, "Management Concepts for Top Down Structured Programming," IBM Corporation, Report No. FSC 73-0001, November 1972, Revised February 1973, 27 pp.

Recent advances in programming technology have simulated an examination of the software development process and programming project management techniques. Many beneficial changes now can be made. This report identifies some of these changes. The concepts of structured programming, top down programming and programming support libraries are reviewed. Testing and management concepts for top down implementation are presented. A documentation strategy is recommended.

M6 H. Mills, "Chief Programmer Team Operation," IBM Technical Report FSC 71-5108, 1971.

N1 I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," Sigplan Notices, V. 8, N. 8, August 1973, pp. 12-26.

With the advent of structured programming and GOTO-less programming a method is needed to model computation in simply ordered structures, each representing a complete thought possibly defined in terms of other thoughts as yet undefined. A model is needed which prevents unrestricted transfers of control and has a control structure closer to languages amenable to structured programming. Presents an attempt at such a model.

N2 Peter Naur, "Proof of algorithms by general snapshots," BIT 6, 1966, pp. 310-316.

A constructive approach to the question of proofs of algorithms is to consider proofs that an object resulting from the execution of an algorithm possesses certain static characteristics. It is shown by an elementary example how this possibility may be used to prove the correctness of an algorithm written in ALGOL 60. The stepping stone of the approach is what is called General Snapshots, i.e., expressions of static conditions existing whenever the execution of the algorithm reaches particular points. General Snapshots are further shown to be useful for constructing algorithms.

N3 Peter Naur, "Programming by action clusters," BIT 9, 1969, pp. 250-258.

The paper describes a programming discipline, aiming at the systematic construction of programs from given global requirements. The crucial step in the approach is the conversion of the global requirements into sets of action clusters (sequences of program statements), which are then used as building blocks for the final program. The relation of the approach to proof techniques and to programming languages is discussed briefly. This paper may be regarded as a continuation of the work in several recent papers concerned with techniques for establishing the correctness of algorithms. It combines the constructive approach advocated by Dijkstra, and the proof techniques described by Floyd and Naur. Very briefly, the essential ideas are to develop a technique for constructing algorithms which takes the global requirements of that algorithm as its starting point, and to justify this approach on the basis of the general snapshots needed to prove the algorithm.

P1 D. L. Parnas, "A technique for software module specification with examples," CACM, V. 15, N. 5, May 1972, pp. 330-336.

This paper presents an approach to writing specifications for parts of software systems. The main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal. The technique is illustrated by means of a variety of examples from a tutorial system.

Comment: Complete, precise specifications to program from. Read twice.

P2 D. L. Parnas, "On the criteria to be used in decomposing systems into modules," CACM, V. 15, N. 12, December 1972.

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositons have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Comment: Must reading.

P3 W. W. Peterson and T. Kasami and N. Tokura, "On the capabilities of while, repeat, and exit statements," CACM, V. 16, N. 8, August 1973.

A well-formed program is defined as a program in which loops and if statements are properly nested and can be entered only at their beginning. A corresponding definition is given for a well-formed flowchart. It is shown that a program is well formed if and only if it can be written with if, repeat, and multi-level exit statements for sequence control. It is also shown that if, while, and repeat statements with single-level exit do not suffice. It is also shown that any flowchart can be converted to a well-formed flowchart by node splitting. Practical implications are discussed.

P4 T. W. S. Plum and G. M. Weinberg, "Teaching structured programming attitudes, even in APL, by example," Proc. Fourth Symposium on Computer Science Education, SIGCSE, February 1974.

As a programming assignment in a graduate programming course, students were to program an interactive word game, JOTTO. The language used was APL, under constraints of well-structured programming and complete control of the user-machine interaction. In response to complaints that teamwork was an impediment to programming and that it was not possible to write efficient well-structured programs in APL, the instructors undertook to complete the assignment working as a team. The results of the effort were carefully documented, including experiences with program modification, and are presented here, as they were to the class, to illustrate the principles that should be communicated to professional programmers.

S1 J. T. Schwartz, "Semantic and syntactic issues in programming," Bulletin of the American Mathematical Society, V. 80, N. 2, March 1974.

Written for mathematicians not working in computer field. Very nicely done.

S2 Randall F. Scott and Dick B. Simmons, "Programmer productivity and the Delphi technique," Datamation, V. 20, N. 5, May 1974.

According to this panel, the use of structured or GOTO-less programming is far less important than good documentation, programming tools, and experience.

Comment: A report which down plays the role of goto-less programming on programming productivity. Based on results of Delphi probe of programming project managers.

S3 Stephen W. Smaliar, "On structured programming," CACM, V. 17, N. 5, May 1974, p. 294.

This forum article suggests that some advocates of structured programming are unrealistic, especially those who abuse FORTRAN. He suggests that structured programming may be a fad and describes three "commandments" that are claimed to be applicable to FORTRAN programming.

S4 M. F. Smith, "Structured projects simplify development efforts," Computerworld, June 12, 1974, p. 14.

Recent articles on structured programming indicate a breakthrough in coding techniques, simplifying software systems development projects. To what can we attribute this success? Project participants cite chief programmer team, top-down development, structured programming and development support library as elements of success. While most participants tend to emphasize structured programming as the dominant aspect of success, he sees an even more exciting concept--structured projects.

S5 S. W. Smoliar, "On Structured Programming," ACM Forum, CACM, V. 17, N. 5, May 1974, p. 294.

This letter describes structured programming as a passing fad. The author attempts to extract three basic common-sense rules which summarize and maintain something of lasting value for the programmer. (See G3.)

T1 Ted Tenny, "Structured programming in FORTRAN," Datamation, V. 20, N. 7, July 1974.

Better languages can be written for structured programming, but the industry's investment in FORTRAN will keep it around awhile. For now, here's what to do.

T2 D. Tsichritzis and A. Ballard, "Software reliability," INFOR, V. 11, N. 2, June 1973.

Their approach assumes that there is increasing interest in both practical and theoretical aspects of the reliability of computer software, and this paper reviews many aspects of software design and production which affect reliability. For the most part, the topics are discussed relative to simple examples, and with reference to the previous work of others; however, a new approach to formally proving system correctness is presented. The system can be represented at any instance of time by its state. The progress of the system is represented by a "state history." Any property can therefore be described as a relation between states. The correctness proof is an induction with respect to the sequence of such states followed during execution. The paper also covers, in review, program design, protection, programming style, testing and other topics.

T3 D. Tsichritzis, "Beautiful systems programming concepts," INFOR, V. 10, N. 1, February 1972.

Concepts enable the designer to understand large operating systems, so that he may design and implement such systems. Four concepts are outlined and their usefulness discussed: Activation Records, Processes, Naming-Binding and Protection Domains.

W1 J. Weizenbaum, "On the impact of the computer on society: how does one insult a machine?", Science, V. 176, N. 12, May 1972, pp. 609-614.

Comment: Must reading. Discusses complexity and the computer as metaphor.

W2 Niklaus Wirth and C. A. R. Hoare, "A contribution to the development of ALGOL," CACM, V. 9, N. 6, June 1966.

A programming language similar in many respects to ALGOL 60, but incorporating a large number of improvements based on six years' experience with that language, is described in detail. Part I consists of an introduction to the new language and a summary of the changes made to ALGOL 60, together with a discussion of the motives behind the revisions. Part II is a rigorous definition of the proposed language. Part III describes a set of proposed standard procedures to be used with the language, including facilities for input/output.

W3 N. Wirth, "The design of a PASCAL compiler," Software-Practice and Experience, V. 1, 1971, pp. 309-333.

The development of a compiler for the programming language PASCAL is described in some detail. Design decisions concerning the layout of program and data, the organization of the compiler including its syntax analyser, and the over-all approach to the project are discussed. The compiler is written in its own language and was implemented for the CDC 6000 computer family.

W4 N. Wirth, "The programming language Pascal," Acta Informatica, V. 1, N. 1, 1971, pp. 35-63.

The development of the language Pascal is based on two principal aims: to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language, and to develop implementations of this language which are both reliable and efficient on presently available computers. The main extensions relative to Algol 60 lie in the domain of data structuring facilities, since their lack in Algol 60 was considered as the prime cause for its relatively narrow range of applicability. The introduction of record and file structures should make it possible to solve commercial type problems with Pascal, or at least to employ it successfully to demonstrate such problems in a programming course. The syntax of Pascal is summarized in graphical form in the Appendix.

W5 Niklaus Wirth, "Program development by stepwise refinement," CACM, V. 14, N. 4, April 1971.

> The creative activity of programming (to be distinguished from coding) is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.
>
> Comment: Must reading.

W6 Ray W. Wolverton, "The cost of developing large-scale software," IEEE Transactions on Computers, V. C-23, N. 6, June 1974.

> The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. Resources (cost) are allocated over the project's life cycle from the structural forecasts. Judgment, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a 25 X 7 structural forecast matrix that has been used by TRW with good results over the past few years is presented in this paper. With the structural forecast in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be based on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five tradition methods of software cost forecasting are described: top-down estimating, similarities and differences estimating, ratio estimating, standards estimating, and bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented that illustrate the software cost forecasting methodology and historical results.
>
> Comment: Very long but contains valuable information.

W7 John D. Woolley and Leland R. Miller, "LINUS: A structured language for instructional use," Proc. Fourth Symposium on Computer Science Education, February 1974.

> One of the crucial decision in organizing a first course in computer science is the choice of a programming language. Although there is considerable variance of opinion as to what the ideal language should be, two main approaches can be delineated. The first approach stresses the necessity of learning the dominant scientific language, which in the Americas amounts to a vote for Fortran. The practicality of this choice is as indisputable as the awkardness of the syntax of that language. The alternative view stresses the importance of the program structure in developing a sound sense of "algorithmic thinking." Proponents of this view would suggest Algol W or perhaps Pascal. The authors contend that both approaches have important advantages. This paper explores an approach which attempts to maximize the benefits of both. The solution they have adopted is to implement a language called Linus (Language for instructional use). This language is pre-processed to ANS Fortran, but has more the appearance of PL/I or Algol 68, facilitating learning corresponding features of those languages. The majority of the language has been implemented and is presently undergoing testing.

W8 William A. Wulf, "Programming without the goto," Proc. IFIP Congress 71, Ljubljana, August 1971.

> It has been proposed, by Dijkstra and others, that the use of the goto statement is a major villain in programs which are difficult to understand and debug. The proponents of eliminating the goto contend that when it is eliminated the resulting program structure admits a simple, systematic proof of correctness. This suggestion has met with skepticism in some circles. This paper analyzes the nature of control structures which cannot be easily synthesized from simple conditional and loop constructs. This analysis is then used as the basis for developing the control structures of a particular language, Bliss. The results of two years of experience programming in Bliss, and hence without goto's, are summarized.

W9 W. A. Wulf, D. B. Russel, A. N. Habermann, "BLISS: A language for systems programming," CACM, V. 14, N. 12, December 1971.

A language, BLISS, is described. This language is designed so as to be especially suitable for use in writing production software systems for a specific machine (the PDP-10): compilers, operating systems, etc. Prime design goals of the design are the ability to produce highly efficient object code, to allow access to all relevant hardware features of the host machine, and to provide a rational means by which to cope with the evolutionary nature of systems programs. A major feature which contributes to the realization of these goals is a mechanism permitting the definition of the representation of all data structures in terms of the access algorithm for elements of the structure.

Comment: The language has no goto but provides for exit from a control statement.

W10 William A. Wulf, "A case against the GOTO," Sigplan Notices, V. 7, N. 11, November 1972.

It has been proposed, by E. W. Dijkstra and others, that the goto statement in programming language is a principal culprit in programs which are difficult to understand, modify, and debug. More correctly, the argument is that it is possible to use the goto to synthesize program structures with these undesirable properties. Not all uses of the goto are to be considered harmful; however, it is further argued that the "good" uses of the goto fall into one of a small number of specific cases which may be handled by specific language constructs. This paper summarizes the arguments in favor of eliminating the goto statement and some of the theoretical and practical implications of the proposal.

Y1 Edward Yourdon, "A brief look at structured programming and top-down program design," Modern Data, June 1974.

> Never before has a programming development stirred as much interest and controversy as the topic discussed in this article. Whether or not you are a programmer, structured programming--like virtual memory or microprogramming--is too important a technique to ignore.

Z1 Ch. T. Zahn, "A control statement for natural top-down structured programming," Proc. Colloque Sur la Programmation, Paris, April 1974.

Comment: A very interesting control structure. The statement form is

until E_1 or E_2 or ... Endo S case of Begin
E_1 : S_1; ... En:Sn end

See also Knuth, K3.

Z2 M. V. Zelkowitz, "It is not time to define structured programming," Operating Systems Review, V. 8, N. 2, April 1974, pp. 7-8.

A response to Denning's letter. Chooses to identify programming as software engineering with the basic phases of design, implementation and testing.

Supplementary Listing

SB1 F. T. Baker, "System quality through structured programming," Proc. FJCC, 1972, pp. 339-343.

Comment: 25-50 bugs in 80,000 lines of code (on time). Significant and impressive case for structured programming.

SB2 R. M. Balzer, "On the future of computer program specification and organization," ARPA Report 622 Rand, Santa Monica, Calif., August 1971.

SB3 P. Brinch-Hansen, Operating System Principles, Englewood Cliffs, Prentice-Hall, 1973.

Comment: Excellent.

SC1 D. C. Cooper, "Reduction of programs to a standard form by graph transformation," Theory of Graphs, International Symposium, Rome, 1966, (Ed. Rosenstiehl, P.), Gordon and Breach, New York, 1967.

SC2 D. C. Cooper, "On the equivalence of certain computations," Computer Journal 9, 1966, pp. 45-52.

SC3 D. C. Cooper, "Bohm and Jacopini's reduction of flow charts," Letter to the Editor, CACM V. 10, August 1967.

Comment: Must reading. See remarks in K3.

SC4 R. Conway and D. Gries, An Introduction to Programming - A Structured Approach Using PL/I and PL/C, Cambridge, Mass., Winthrop Publishers, 1973.

SD1 O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press, New York, 1972.

Program design by Dijkstra, Data Structuring by Hoare, Hierarchical Program Structures by Dahl and Hoare fairly heavy reading, but well worth the effort. Read repeatedly.

SD2 E. W. Dijkstra, "Go to statement considered harmful," Letter to the Editor, CACM, V. 11, March 1968.

Comment: Perhaps first article on structured programming.

SD3 E. W. Dijkstra, "A short introduction to the art of programming," Report 316, Technische Hogeschool Eindhoven, August 1971.

SD4 E. W. Dijkstra, "Concern for correctness as a guiding principle for program composition," The Fourth Generation, Infotech, Ltd., Berkshire, England, 1971, pp. 347-367.

SD5 E. W. Dijkstra, "Programming considered as a human activity," Proc. IFIP Congress 65,65, edited by W. A. Kalenich, Spartan Books, Washington, D. C. 1965.

SF1 R. W. Floyd, "Assigning meanings to programs, Proc. Symposium Applied Math., AMS, V. 19, 1967.

SG1 B. Galler and A. Perks, A View of Programming Languages, Reading, MA, Addison Wesley, 1970.

SH1 W. C. Hetzel (ed.), Program Test Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

SI1 Y. I. Ianov, "On the equivalence and transformation of program schemes," CACM V. 1, 1958, pp. 8-12.

SJ1 J. B. Johnston, "The contour model of block structured processes," Proc. Symposium on Data Structures in Programming Languages, Sigplan Notices, V. 6, N. 2, February 1971.

SL1 P. J. Landin, "The next 700 programming languages," CACM, V. 9, March 1966.

SM1 R. C. McHenry, "Management concepts for top-down structured programming," IBM Technical Report No. FSC-73-0001, February 1973.

SM2 E. F. Miller, Jr., A Compendium of Language Extensions to Support Structured Programming, General Research Corp., RN-42, January 1973.

SM3 H. Mills, "The case against GOTO statements in P1/1," IBM Report No. C224H2, April 1969.

SR1 J. R. Rice, "The goto statement reconsidered," Letter to the Editor, CACM, V. 11, 1968, p. 538.

SW1 G. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold Company, New York, 1971.

Lightly written and fascinating to any programmer.

Comment: Interesting reading.

SW2 N. Wirth, "On certain basic concepts of programming languages," Computer Science Technical Report No. CS65, Stanford University, 1967.

SW3 N. Wirth, Systematic Programming An Introduction, Englewood Cliffs, Prentice-Hall, 1973.