AN ANNOTATED BIBLIOGRAPHY ON STRUCTURED PROGRAMMING

Compiled

by

W. R. Franta

UCC TR 74-1

PREFACE

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 document presents an annotated bibliography of papers and articles on the subject. As stated by Webster, to annotate means to provide explanatory notes. At first draft the annotations were my own, and owing to personal bias were never, to my mind, sufficiently objective to bear passing along. A second draft was therefore undertaken to produce a version in which much of my personal assessment of the listing was replaced by the abstracts provided by the original authors. In cases where I remain compelled to comment my remarks are clearly labeled and can, therefore, be easily ignored.

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 I have included listings of seminal as well as "bandwagon" works. No attempt has been made to label either as both types can contribute to our 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 I am 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 without abstract or comment of several items. The latter are either books, the summaries of documents which require more than a single paragraph of annotation or items unavailable (to me) for inspection.

Finally, I apologize now to the many authors whose works are 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.

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, coroutines, critical sections, distributive operators, dynamic instruction modification, expressions, generators, implicit coroutines, implicit sequencing, iterative control, indivisibility, interleaved execution, the go to, 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 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.

F3 Clinton R. Faulk, "Yet another attempt to define "structured programming," Operating Systems Review, V. 8, N. 3, July 1974.

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

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.

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 conventioanl iterative constructions ('for," "while," etc.) are more readable and more easily proved correct. In this note we will make a few exploratory observations 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 July 1974.

> 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.

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," <u>BIT</u>, 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.

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.

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.

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 micro-programming--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

$\underline{\text{until}}\ E_1\ \underline{\text{or}}\ E_2$ or ... End$\underline{\text{do}}$ S $\underline{\text{case}}$ $\underline{\text{of}}$ $\underline{\text{Begin}}$

$E_1 : S_1;$ ... En:Sn $\underline{\text{end}}$

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/1 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.

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 Compendum of Language Extensions to Support Structured Programming, General Research Corp., RN-42, January 1973.

SM3 H. Mills, "The Case against GOTO statements in Pl/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.