LBL-3004
c.2

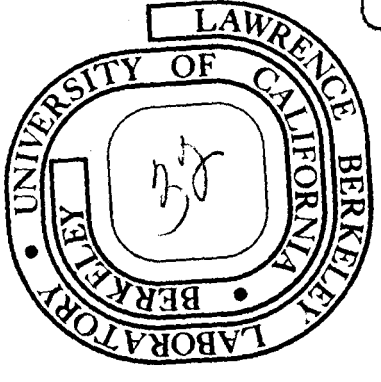
To be presented at the Annual Conference of the Association for Computing Machinery, San Diego, Ca., November 11-13, 1974.

A METHOD TO EXPOSE THE HIDDEN STRUCTURE OF FORTRAN PROGRAMS

Loren P. Meissner

April 22, 1974

Prepared for the U. S. Atomic Energy Commission
under Contract W-7405-ENG-48



LBL-3004
c.2

DISCLAIMER

[SIGPLAN or SIGCSE]

Loren P. Meissner
Lawrence Laboratory
University of California
Berkeley, Calif. 94720

(415-843-2740)

A METHOD TO EXPOSE THE HIDDEN STRUCTURE OF FORTRAN PROGRAMS

Abstract

Program structure is inherent in program design; therefore special keywords such as "if ... then ... else" or "do ... while" are useful only to the extent that they *reveal* that structure.

A simple listing of Fortran program statements is ineffective for revealing program structure. Proposals have been made for manually inserting keywords, comments, indentations, etc., either during a separate preprocessing stage or during the normal coding process.

We show how the *flow graph* can provide independent structural information. Although the flow graph may be said to exist as soon as a program has been designed, it is most readily generated from the program statements. "Bad" structure can be detected objectively, and "good" programs can be reconstituted to reveal their block structure more clearly. Our implementation is based on an algorithm suggested by Peterson et al (CACM, August 1973). We have extended this algorithm to automatically detect block exits.

Background

In recent years a great deal of discussion [1] - [14] has centered around the structure of computer programs, particularly as this structure is reflected in the flow of control of the program during execution. In much of this discussion, however, one point has perhaps received inadequate attention. It has not been strongly enough emphasized that the structure of a program is pretty firmly established during the program design phase, and therefore that not much can be done during the coding phase to change or to augment program structure.

There is little doubt that when the designer of a program thinks of it as being composed of *sequences*, *alternatives*, and *repetitions*, the resulting program will generally be easier to comprehend (by the designer or by others later on), will be easier to prove correct, and will be easier to compile (and especially to optimize). But excessive attention has been devoted to the coding phase, with excessive emphasis on the use of particular keywords ("if ... then ... else" and "do ... while") and the implication that program structure resides in these keywords. It has sometimes been forgotten that structure is inherent in the program design, and that keywords are useful only to the extent that they *reveal* structure which already exists.

The Fortran user's dilemma. The difficulty with Fortran programs, from this point of view, is not that they lack structure but that their structure may be difficult to discern. The keywords that appear in the control statements of a Fortran program do little to enhance the recognition of program structure; indeed, in most instances (perhaps with the exception of "do") they tend to obscure it. Because of the lack of such keywords as "if ... then ... else" and "do ... while," Fortran users are forced to use the keyword "go to" in a variety of ways, to implement many different constructs. This makes the structure of a Fortran program difficult to recognize from a listing of the program statements.

For a Fortran user who is convinced of the benefits of structured programming, is there any choice but to switch to some other language? Although he concedes that Fortran is far from ideal, he may feel that most other existing languages are not much better [15]. He would like to continue using Fortran, and yet he would like to adhere to the principles of structured programming.

Software aids should be developed to help the Fortran program designer understand the structure of his programs. A particular objective should be to help him distinguish in some manner between the different uses of "go to." Even in Dijkstra's letter [2], the point is made that the "go to" *per se* is not so "harmful" as its "unbridled use." It should be possible to find an objective way of distinguishing between those uses of "go to" that are unbridled and therefore harmful, and those on the other hand that are bridled and hence benign.

Extensions to Standard Fortran. One possible way to gain some of the advantages of structured programming would be to add the keywords "if ... then ... else" and "do ... while" to Fortran. Several experiments have been made in this direction, including "structured Fortran" extensions named DEFT [16], IFTRAN [17], LINUS [18], MORTRAN [19], and SPIFFY [20]. (No doubt there are others.)

The obvious approach is to implement a Fortran extension as a preprocessor, which accepts programs written in a "structured" language more or less resembling Fortran, and translates them into Standard Fortran programs which can then be processed in the ordinary way. However, the preprocessor approach introduces an additional language level during debugging, and therefore these implementations may prove to be important principally as experimental tools or testing grounds for the evaluation of the Fortran extension ideas they embody. Meanwhile, further experiments of this nature should certainly be encouraged, and wide dissemination of various proposals (along with reports of experiences of their users) should be promoted.

Informal techniques. Other proposals involve the use of an informal language to develop a pseudo-program which the programmer then translates by hand into Standard Fortran. One such proposal is the "Programming Design Language (PDL)" [21]. The control structures are expressed using keywords such as "if ... then ... else" along with systematic indentation, while the statements controlled by these keywords are written out in plain English. In principle, such a pseudo-program could be translated with equal ease into Fortran, Cobol, Algol, PL/1, or any other language.

T. E. Hull [22] proposes that comment cards be manually inserted into Standard Fortran programs in certain prescribed ways, e.g., to insert keywords such as "if ... then ... else" or to mark the beginning and end of each block of statements in an alternative clause or a repetitive clause. Furthermore, Hull would restrict the use of "go to" statements to those ways that are necessary in implementing structured programming constructs. However, it is not clear whether the use of "correct" structural principles can be adequately enforced in a manual system of this kind.

Using flow graph information to expose program structure

A simple listing of Fortran program statements is not adequate for revealing program structure. The extensions and informal techniques described above attempt to correct this deficiency with auxiliary information that is inserted manually in the listing, either during a separate preprocessing stage or during the normal coding process.

We propose to tap an independent source of structural information, and to make the program *flow graph* available to the user. Although this information exists (in a sense) as soon as the program has been designed, it is most readily captured by the computer after the program statements have been written. We have implemented some software which will scan a Fortran program or subprogram, and will generate and display its flow graph. The flow graph is also used to produce a restructured listing of the program statements, as illustrated in Fig. 1 (which is based on an example discussed by Hull [22]). The techniques described in the remainder of this paper are based on a synthesis and extension of the studies and proposals of Hecht and Ullman [23] and of Peterson, Kasami, and Tokura [24]. (These two papers are hereinafter referred to as HU and PKT, respectively.)

In the current preliminary version of our program, a node of the flow graph corresponds to each:

labelled statement (except "format" statements),
"go to" statement (including computed "go to"),
"if" statement (logical or arithmetic),
"do" statement,
"stop," "return," or "end" statement.

An arc of the flow graph leads from each node to those nodes which can immediately follow it in the execution sequence.

Well structured program flow graphs. A flow graph corresponding to a program that is composed entirely of sequences, alternative clauses, and iterative clauses, is called a *"D-chart."* Flow graphs of this form have been studied extensively [25], [26]. It has been proved that the flow graph of an arbitrary program can be reduced to a D-chart; however, this reduction may increase the length of a program or alter its execution sequence.

The term "D-chart" includes flow graphs corresponding to programs whose alternative clauses may include more than two branches (e.g., "case" clauses). However, the test for completion of an iteration clause must be made at the beginning of the loop[1]. In a D-chart, every subgraph has one entry and one exit. Thus the flow graph of the program in Fig. 1 is not a D-chart. One way to translate this program to the "do ... while" form is to include one additional execution of the assignment *RN = RN + 1.0* , after the variable *TERM* has already reached its final value. Knuth and Floyd [5], Ashcroft and Manna [6], and PKT all give examples of programs whose flow graphs cannot be reduced to D-charts without some essential (although perhaps minor) modification. Such changes, motivated by a desire to force all programs into the D-chart mold, may obscure rather than clarify the inherent program structure.

Recent discussions of structured programming [14], [27] tend to the concensus that a fourth basic structural unit, the *multi-level exit,* should be permitted in addition to sequences, alternative clauses, and iterative clauses. Experience shows that incorporation of this structural form is generally justifiable from the standpoint of program comprehension, even though any program can, in principle, be re-cast to avoid it. Programs composed from these four structures correspond to the *well formed* flow graphs discussed in PKT.

Accordingly, the term *"well structured"* may be adopted to describe the class of programs which are composed of sequences, alternative clauses, iterative clauses, and multi-level exits. In Fig. 2, the region D represents the set of programs whose flow graphs are D-charts. Programs that require multi-level exits (or some equivalent modification) in addition to the properties of D-charts are represented by the region E. Thus the set D ∪ E comprises the well structured programs (according to this terminology).

PKT gives an alternate characterization of the flow graphs of programs in the set D ∪ E. It is shown that a program can be composed entirely from these four structural units, if and only if its flow graph does not contain

- - - - - -

[1] It is immaterial to this discussion whether the test is also made prior to the first iteration ("do ... while"), or only prior to iterations after the first ("do ... until").

any strongly connected[2] subgraph with more than one entry node. It does not seem unreasonable to exclude from the class of well structured programs one whose flow graph contains some strongly connected subgraph with more than one entry node, and which therefore cannot be composed from the four "permissible" structural units[3].

HU shows that a program flow graph is *reducible* (in a certain sense which is important for program verification and especially for optimization) if and only if it contains no strongly connected subgraph with more than one entry node, and that the D-charts form a subset of the reducible flow graphs. Thus the set of programs having reducible flow graphs also corresponds exactly to the set D ∪ E of Fig. 2. HU also shows that any Fortran program whose transfers to previous statements are all caused by the normal termination of "do" loops is reducible.

A comparison of HU and PKT suggests an objective criterion for distinguishing between correct and incorrect uses of the "go to" statement in Fortran programs. In a well structured program, such a statement may be used to implement a "downward" flow of control, or even to produce an "upward" or backward flow in a manner that is equivalent to the normal flow of control in a "do" loop. An improper use of the "go to" statement, on the other hand, would be one which introduces more than one entry into a strongly connected subgraph of the program flow graph. (Our algorithm does not proceed with the flow graph analysis after it finds such an "improper" "go to," but instead it returns the program, along with some flow graph information, to the originator for correction.)

Conversion of well structured flow graphs to nested form. In a well structured program flow graph, each strongly connected subgraph has a unique entry node. If we delete all *return arcs* (arcs leading to the entry node of a strongly connected subgraph, from within the subgraph), the resulting graph will have no strongly connected components. Nevertheless, it may contain subgraphs of "hammock" form, corresponding to alternative clauses in the program. That is, there may be a pair of nodes (such as nodes 2 and 8 in Fig. 3) that are joined by more than one path in the same direction. PKT shows how to arrange the nodes of such a graph (having no strongly connected subgraphs)

- - - - - -

2 A *strongly connected* subgraph is one which has the property that between any two of its nodes i and j there is at least one path (sequence of arcs) leading in each direction, i.e., from i to j and from j to i. An *entry node* of a subgraph is a node in the subgraph that is the endpoint of an arc originating outside the subgraph.

3 PKT shows how to correct a program that is not well structured, by using a transformation called *"node splitting."* This is a way of preserving one entry node of each strongly connected subgraph and removing all the others. Each entry node to be removed is duplicated, along with that portion of the subgraph connecting it to the remaining entry node. Programs whose flow graphs can be reduced by node splitting to D-charts correspond to the region F in Fig. 2. Any program flow graph can be reduced by node splitting to the graph of a program in the set D ∪ E.

into a single linear sequence or vector, in such a manner that no arc goes from any node to another node that precedes it in this sequence. Thus all flow is "downward" except for the return arcs that have been temporarily removed from consideration.

The key to this algorithm (and, incidentally, the most complex part from the computational standpoint) involves the discovery of the *lowest cover* for each *merge node*. [A merge node is any node with more than one arc leading to it; and its lowest cover is the "lowest" node (in the sense that arcs of the flow graph are directed "downward") through which every path passes that leads to the merge node.] Whenever it is discovered that the next node (to be chosen as an element of the vector) is the lowest cover for some merge node, that merge node is pushed onto a stack. It is shown in PKT that this guarantees no node will be included twice in the vector. On the other hand, when a node in the vector has all its successors already on the stack, the next node is obtained by popping the stack.

This use of an auxiliary pushdown stack induces an implicit *nesting* relationship among the nodes of the flow graph. The level of nesting of a node may be defined to be the depth of the stack at the time the node was placed in the vector. Pushing a node on the stack increases the nesting depth, and therefore begins a subsequence of nodes that are all at (or deeper than) a certain nesting level. This subsequence is, in effect, a *block* within the flow graph, corresponding to a block of program statements. This block ends at the point where the node is popped from the stack and incorporated in the vector.

Automatic detection of exit arcs. The procedure described so far is based closely upon the algorithm described in PKT. However, some experience with this procedure drew our attention to an anomalous result. When the exit from a loop forms the only path to a certain part of the program, the information contained in the program flow graph makes it appear that that part of the program belongs entirely inside the loop. For example, consider the interpolation program illustrated in Fig. 3. A loop is used to search for a certain value of an index, and when an appropriate index value is found, control exits from the loop and the entire remainder of the program is then executed. The PKT algorithm incorporates virtually the entire program within the loop.

We have extended the algorithm to detect arcs that exit from a loop. The strongly connected subgraphs (identified by their entry nodes) form a tree. An exit arc may be defined as an arc leading from one strongly connected subgraph to another that is not contained within it. We find the outermost strongly connected subgraph that contains the source node of the exit arc but does not contain its destination node. Before the entry node of this subgraph is placed in the vector, the destination node of the exit arc is pushed onto the stack, thus creating an additional block level. The resulting nested flow graph is shown in Fig. 4.

Reconstituting the program listing

We use the results of this flow graph analysis to produce a restructured version of the original program (see Fig. 1). Nesting levels are displayed

by means of successive indentations. Comments are inserted to mark the beginning and end of each block, the statements from which flow returns to a loop entry point, and the statements causing control to exit from a block.

Remarks

This same technique can, in principle, be applied to programs written in languages other than Fortran; the main requirement would be a simple adaptation of the process of generating the flow graph from the program statements. However, it is the Fortran language that seems most clearly to be in need of automatic aids to program structure recognition.

We have implemented this technique initially as a post-processor (which operates upon programs that have already been compiled, and thus may be assumed to be free from syntax errors). However, certain advantages would accrue from its incorporation as an integral (presumably optional) part of a compiler. Most compilers already have some reasonable equivalent of the program flow graph available; and conversely, the results of our analysis should be of value in code optimization.

Experience of users. We have applied this algorithm to a few programs written by experienced programmers whose habits are probably rather conservative. We often found that the nodes of the linearized flow graph corresponded to the statements of the program in their original sequence, and that most of the exceptions resulted from the arbitrary selection between a pair of arcs emanating from a single node.

The most common case of "ill structure" that was found in this limited sample consisted of "exception" processing applied during execution of a loop. For example, during a loop to process the characters of an input string, a special sequence of statements is executed when the end of a card is reached. If this same exceptional condition can occur before the loop is entered (e.g., while searching for the beginning of an input string), even a fairly conservative programmer may succumb to the temptation to code a jump to the processing segment inside the loop.

Bibliography

1. C. Bohm and G. Jacopini "Flow diagrams, Turing machines, and languages with only two formation rules" Comm. ACM 9:3, 1966
2. E. W. Dijkstra "Go to statement considered harmful" Comm. ACM 11:3, 1968
3. R. M. Burstall "Proving properties of programs by structural induction" Computer Journal 12:1, 1969
4. D. E. Cooper "Programs for mechanical program verification" Machine Intelligence 6, 1971
5. D. E. Knuth and R. W. Floyd "Notes on avoiding go to statements" Information Processing Letters 1, North-Holland, Amsterdam, 1971
6. E. Ashcroft and Z. Manna "The translation of go to programs into while programs" Proceedings IFIP-71, North-Holland, Amsterdam, 1972
7. F. T. Baker "System quality through structured programming" Proceedings AFIPS 1972 Fall Joint Computer Conference, Vol. I, 1972

Bibliography *(cont.)*

8. E. W. Dijkstra "Notes on structured programming" in Structured Programming by Dahl, Dijkstra, and Hoare, Academic Press, 1972
9. E. W. Dijkstra "The humble programmer" Comm. ACM 15:10, 1972
10. C. A. R. Hoare "A note on the for statement" Bit 12:3, 1972
11. M. C. Hopkins "A case for the go to" SIGPLAN Notices 7:11, 1972
12. B. M. Leavenworth "Programming with(out) the go to" Proceedings ACM 1972; SIGPLAN Notices 7:11, 1972
13. H. D. Mills "How to write correct programs and know it" IBM Report, Gaithersburg, Md., 1972
14. W. A. Wulf "A case against the go to" Proceedings ACM 1972; SIGPLAN Notices 7:11, 1972
15. A. Ralston "The future of higher level languages (in teaching)" Proceedings International Computing Symposium 1973, North-Holland, Amsterdam, 1974
16. C. A. Steele and A. E. Sedgwick "DEFT: A disciplined extension of Fortran" Tech. Report, Dept. of Computer Science, U. of Toronto, 1973
17. E. F. Miller "Extensions to Fortran to support structured programming" SIGPLAN Notices 8:6, 1973
18. L. Miller "LINUS: A structured language for instructional use" SIGCSE Bulletin 6:1, 1974
19. A. J. Cook "A user's guide to CDC 7600/6600 MORTRAN" TM 150.1, Computation Group, Stanford Linear Accelerator, Stanford, 1973
20. L. Carpenter *(private communication)*, Boeing Computer Services, Inc., Seattle, 1974
21. "Improved technologies for applications development" (Report), Productivity Techniques Dept., IBM Corp., Bethesda, Md., 1973
22. T. E. Hull "Would you believe structured Fortran?" SIGNUM Newsletter 8:4, 1973
23. M. S. Hecht and J. D. Ullman "Flow graph reducibility" Soc. Ind. Appl. Math., Journal on Computing 1:2, 1972
24. W. W. Peterson, T. Kasami, and N. Tokura "On the capabilities of while, repeat, and exit statements" Comm. ACM 16:8, 1973
25. J. Bruno and K. Steiglitz "The expression of algorithms by charts" TR 88, Computer Science Laboratory, Princeton U., 1971
26. D. C. Cooper "Some transformations and standard forms of graphs, with applications to computer programs" Machine Intelligence 2, American Elsevier, N.Y., 1968
27. G. V. Bochmann "Multiple exits from a loop without the go to" Comm. ACM 16:7, 1973

```
                    *C      start
1  → (2)                    EPS = 1.0E-5
                    *C      L10:  begin iterative structure
2  → (10, 3)                   DO 4 I = 1, 11
                    *C      ...
                               X = 1.0 + FLOAT (I - 1) / 10.0
                               TERM = 1.0
                               SUM = 1.0
                               RN = 1.0
                    *C         L5:  begin iterative structure
3  → (4)              1            TERM = - TERM * X / RN
                                   SUM = SUM + TERM
4  → (7, 5)                        IF (ABS (TERM) .GE. EPS) GO TO 3
                    *C         ...
- - - - - -        *C            sequence break
7  → (8)              3            RN = RN + 1.0
8  → (3)↑                          GO TO 1
                    *C            return arc
- - - - - -        *C            sequence break
                    *C         end L5
5  → (6)              2         WRITE (6, 100) X, SUM
                     100       FORMAT (1X, F 4.1, F 10.5)
6  → (9)                       GO TO 4
- - - - - -        *C         sequence break
9  → (2)↑             4         CONTINUE
                    *C         return arc
                    *C      end L10

10 → (11)
11 → (12)                   STOP
12                          END
```

Figure 1. Restructured listing based on the linearized flow graph of a program (adapted from Hull [22]) to compute a table of values of the exponential function for negative arguments. On the left is a representation of the flow graph in numeric form, using *node numbers* that have been assigned consecutively (and do not necessarily agree with statement labels). Comments preceded by an asterisk were generated by the analysis algorithm.

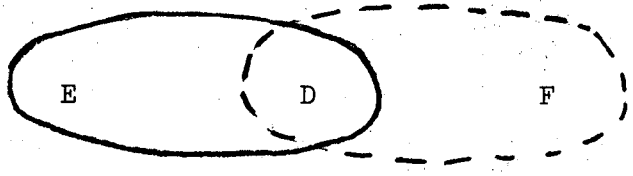


Figure 2. Classification of programs, according to the characteristics of their flow graphs. Flow graphs of programs in set D are "D-charts": that is, they are composed entirely of sequences, alternative structures, and iterative structures. Programs in set E may also include multi-level exit structures. The set D ∪ E contains all "well structured" programs. Programs in set F can be reduced to set D by "node splitting." Any program can be reduced to set D ∪ E by node splitting.

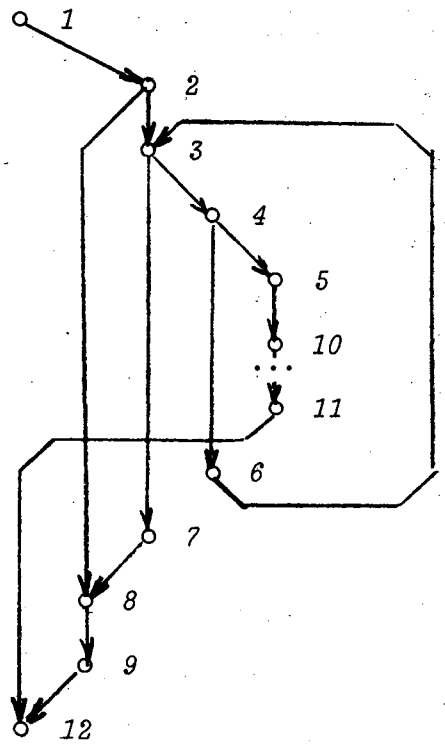


Figure 3. Linearized flow graph of an interpolation subprogram, constructed according to Peterson's algorithm [24]. In an actual application, many more statements might appear between nodes 10 and 11. Note that these statements are included within the iteration structure.

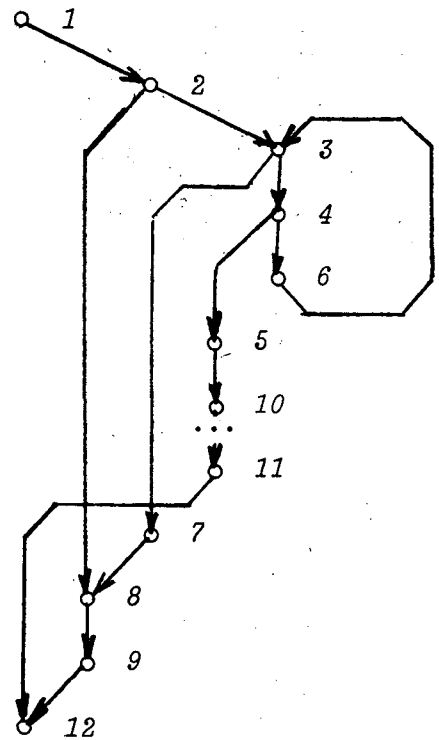


Figure 4. A modified version of the algorithm has been applied to the same flow graph. Note that nodes 10 and 11 have moved outside the iteration structure.

Appendix

Space restrictions prohibited inclusion of the following material in the paper submitted to ACM-72.

On keywords. A rather extreme but widely advocated position is that programmers should use the keywords "if ... then ... else" exclusively when they have a pair of alternatives to express, that they should use the keywords "do ... while" to express every repetition, and that the four-letter word "go to" [11] should never be mentioned in the polite society of the statements of a well structured program.

It has been implied, in fact (chiefly by proponents of PL/1), that the primary criterion for the choice of a programming language should be the appearance of a proper set of keywords among its control statement forms. D. Gries [A] insists that "the only choice" for teaching introductory programming is the PL/1 language, because "it has the compound statement, the conditional statement, and the while loop." D. D. McCracken [B] predicts "a swing to PL/1, precisely because it is well-suited for structured programming." R. C. Holt [C] is less dogmatic, but he lists as one of five reasons for choosing PL/1 the fact that it "has somewhat reasonable control structures (DO-WHILE and IF-THEN-ELSE)."

Use of special keywords, when they are available, does have advantages. It must be admitted that our choice of notation affects the way we think [8]. Use of "do ... while" to control repetitions can emphasize the fact that it is not always necessary to employ an index variable or repetition counter. It also seems sensible to avoid using "go to" where other keywords are available that adequately serve the intended purpose. In PL/1, for instance, the use of "go to" can be restricted to certain exceptional situations, thus drawing attention to these situations as potential trouble spots [11].

On Fortran extensions. It may be observed that the designers of most of these Fortran extensions [16] - [20] have been unable to resist the temptation to go beyond the minimum essentials for making Fortran into a language in which the structure of a program can be readily observed. The extensions often include so many other features that they might better be characterized as PL/1 subsets than as Fortran supersets. Only time will tell whether a widely acceptable extension can be developed which includes the necessary structural features, while remaining close enough to the present language to be properly called Fortran.

Along with the "structured" keywords, it is obviously necessary to introduce into the Fortran language some provision for statement grouping or "block" structure. A key difference between the structure represented by "if ... then ... else" and the Fortran "if" lies in the fact that the conditional clause controlled by the latter can include only one statement. The means of introducing block structure into an extended Fortran language deserves perhaps even more consideration than the set of keywords included.

Books and articles [D] - [F] have been appearing for years exhorting Fortran programmers (as well as the users of other languages) to insert comments among the statements of a program. The newer proposals would tend to make this added information more systematic, and would provide new software aids for its incorporation.

Implementation details

(1.) Nodes (including labelled statements) are numbered consecutively; thus node numbers do not necessarily agree with statement labels. During the input phase, statement labels are stored in alphanumeric form. After the end of the input source program is encountered, the alphanumeric labels are matched during a second pass, and node numbers are substituted.

(2.) When a "do" statement is encountered in the Fortran program being analyzed, our algorithm of course generates a corresponding node. (There will be an arc leading to this node for each explicit flow path leading to the "do" statement, and there will be one arc leading from the "do" to the next consecutive node.) The target label of the "do" is also pushed onto a special stack, along with the node number of the "do" statement that references it. Since several nested "do" statements may have the same target label, this label may appear more than once in consecutive positions on the stack. When the matching labelled statement is encountered, it is removed from the stack and an extra target node is created for each occurrence on the stack. Arcs are generated from the matching "do" node to the created node, and from the node preceding the created node to the matching "do" node. (The search for strongly connected subgraphs will identify return arcs of "do" loops in the normal way. No special strategy is used for finding these particular return arcs, so that a "do" loop with multiple entries can be detected.)

(3.) Each node in the flow graph has either one or two outgoing arcs, except for a node that corresponds to an arithmetic "if" or to a computed "go to." Extra nodes are created for these statements, so that the flow graph can be stored in a $2 \times n$ adjacency matrix, where n is the number of nodes. The first element in the j'th column of this matrix is the node number of the destination node of one outgoing arc from the j'th node; the second element is the node number of the second outgoing arc if there are two, and is zero if there is only one outgoing arc.

During the search for strongly connected subgraphs, Warshall's algorithm is used to form the connection matrix from the adjacency matrix. The connection matrix is stored in the form of a bit map, and specially coded routines are used to manipulate the bits. Whenever a strongly connected subgraph is found, the return arcs are flagged in the adjacency matrix with a minus sign, and these arcs are ignored when Warshall's algorithm is again used to form the new connection matrix.

(4.) We have not found any efficient way to locate the lowest cover of a merge node. The method we use is as follows. First, a subgraph is formed consisting of all nodes that are on paths from the origin to the merge node. Then some explicit path from the origin to the merge node is found (by working downward from the origin, and testing each outgoing arc to see if the connection matrix indicates the presence of a path from its destination node to the merge node.)

We then work backward up this path until we find a node that is not a cover. To this end, a preliminary check is made, comparing the current node on this path with each node in the subgraph that was formed at the first step. If there is any node in the subgraph that is not connected to the current node by a path (in either direction), then the current node is not

a cover. If the current node is not rejected during this check, a new connection matrix is formed for all the nodes in the subgraph with the exception of the current node, and this connection matrix is checked to see whether there would still be a path from the origin to the merge node if the current node were deleted. The first node encountered during this backward search, whose deletion would break all paths from the origin to the merge node, is the lowest cover of the merge node.

(5.) In the original algorithm, the *"stack point"* for each merge node is its lowest cover, which is located after the analysis of strongly connected subgraphs and before the process of forming the linearized flow graph vector begins. In the algorithm as modified to detect exit arcs, this stack point location step is expanded. For each node i in the flow graph, the following procedure is executed:

(a.) If the node i is not a merge node, let l be the source node of the unique arc leading to node i.

(b.) If the node i is a merge node, let l be the lowest cover of i, and in this case also let l be the stack point of i.

(c.) In either case, if the node i is outside the innermost strongly connected subgraph containing l, then find the outermost strongly connected subgraph that contains l but does not contain i. The entry point of that subgraph is the stack point of node i (replacing the stack point assigned at step (b), if any).

It is also necessary to give exit nodes some priority during the process of arranging the several nodes to be stacked at a given point. If the stack point of i was assigned (or changed) at step (c), then node i should be stacked before other nodes that are unrelated to i according to the partial ordering of the flow graph.

(6.) After the stack points (if any) of all the nodes have been found, the nodes are placed one at a time (starting from the program origin node) in the vector that will ultimately contain the program flow graph in linearized form. After a node is placed in the vector, all nodes for which this node is the stack point are pushed onto the stack. (If there is more than one such node, they are arranged so that the lowest such node will be stacked first.) If the current node has two outgoing arcs, and both of them are "eligible" in the sense of being neither return arcs nor arcs leading to nodes already on the stack, then the destination node of one of the eligible arcs is stacked. If one eligible arc remains, its destination node is taken as the next node to be placed in the vector. If no eligible arc remains, the next node is obtained by popping the stack.

Simplifying assumptions. The current preliminary version of our algorithm is based on some assumptions that should be removed in a final working version. For instance, a control statement that extends onto a continuation card is not scanned beyond the end of the first card. Furthermore, such potential control branches as "call" statements with statement label parameters, or "read" statements with end of file branch labels, are ignored. We also exclude assigned "go to" statements.

More basic is the fact that we do not perform a complete syntactic analysis, but merely examine the first four non-blank characters on each source card (beginning from column 7). Tests are made to see whether these characers are any of the following: "FORM", which indicates that a labelled state-

ment is a format statement and should be ignored, "GOTO", "DO" followed by a decimal digit and any fourth character, "STOP", "RETU" ("return"), "END" followed by any fourth character, and "IF(" followed by any fourth character.

A "GOTO" followed by a left parenthesis is assumed to be a computed "go to." After "IF(", we scan to find a matching right parenthesis, and then resume a detailed scan of the next four non-blank characters on the card. If the first of these is a digit, an arithmetic "if" is indicated. We note that the following sequences may occur legally, anywhere in a Fortran program:

IF (*C*) STOP
IF (*C*) RETURN
IF (*C*) GO TO *n*
IF (*C*) GO TO (*n1*, ..., *nk*), *i*
IF (*C*) IF (*E*) *n1*, *n2*, *n3*
IF (*C*) *non-control statement*

The following may occur anywhere except as the terminal statement of a "do" loop:

IF (*E*) *n1*, *n2*, *n3*
STOP
RETURN
GO TO *n*
GO TO (*n1*, ..., *nk*), *i*
DO *n* ...
END

Non-standard implementation features. The program PETER, at Lawrence Laboratory, deviates from Standard Fortran in that it uses full-word masking operations to generate and analyze the connection matrix. Furthermore, the source program characters are packed, 10 characters per word. However, the few modules involved in these operations are independent of the remaining parts of the program, and could easily be replaced to adapt the program for execution elsewhere.

Proposed pedagogical use. We have considered teaching a beginning programming course, basing the first few weeks of instruction (perhaps 25% of the course) on a simple "go to" - free block structured language (a description of which is available from the author). A set of rules, similar to those of Hull [22], for the correct usage of Fortran statements would then be taught, and would be enforced by the incorporation of a flow graph analysis algorithm into the student version of the campus Fortran compiler. We might expect that the students would transfer their experience with the block structured language to the informal design stage of their Fortran programming tasks. Presentation of the control structures of Fortran would then take the form of showing how to use the available statements of the Fortran language to implement the structures that they had learned to use without the "go to."

Additional references

A. D. Gries "What should we teach in an introductory programming course?" SIGCSE Bulletin 6:1, 1974

B. D. D. McCracken "Revolution in programming: an overview" Datamation 19:12, 1973

C. R. C. Holt "Teaching the fatal disease, or, introductory computer programming using PL/I" SIGPLAN Notices 7:11, 1972

D. G. L. Perry and J. T. Somerfeld "Fortran programming aids" Software Age, Oct - Nov 1970

E. C. B. Kreitzberg and B. Shneiderman "The elements of Fortran style: techniques for effective programming" Harcourt Brace Jovanovich, N. Y., 1972

F. D. D. McCracken and G. M. Weinberg "How to write a readable Fortran program" Datamation 18:10, 1972

G. D. D. McCracken "Is there a Fortran in your future?" Datamation 19:5, 1973

```
      EPS = 1.0E-5
      DO 4 I = 1, 11
      X = 1.0 + FLOAT (I - 1) / 10.0
      TERM = 1.0
      SUM = 1.0
      RN = 1.0
1     TERM = - TERM * X / RN
      SUM = SUM + TERM
      IF (ABS (TERM) .GE. EPS) GO TO 3
2     WRITE (6, 100) X, SUM
100   FORMAT (1X, F 4.1, F 10.5)
      GO TO 4
3     RN = RN + 1.0
      GO TO 1
4     CONTINUE
      STOP
      END
```

Figure A. Fortran program example, adapted from Hull. Computes a table of values of the exponential function with negative arguments.

```
[FOR] X = 1.0 [BY] 0.1 [TO] 2.0

   [  [TERM, SUM, RN] = 1.0;

      [WHILE] ABS (TERM) .GE. EPS

         [  TERM = - TERM * X / RN;

            SUM = SUM + TERM;

            RN = RN + 1.0; ]

      [W] X, SUM; (1X, F 4.1, F 10.5); ]

STOP; END
```

Figure B. The program shown in Fig. A, as it might be rewritten in the "extended" Fortran language *Mortran*.

```
      EPS = 1.0E-5
      DO 4 I = 1, 11
         X = 1.0 + FLOAT (I - 1) / 10.0
         TERM = 1.0
         SUM = 1.0
         RN = 1.0
C        REPEAT
 1          TERM = - TERM * X / RN
            SUM = SUM + TERM
            IF (ABS (TERM) .LT. EPS) GO TO 2
               GO TO 3
C           THEN
 2             WRITE (6, 100) X, SUM
 100           FORMAT (1X, F 4.1, F 10.5)
C         ......EXIT
               GO TO 4
C           ELSE
 3          RN = RN + 1.0
            GO TO 1
 4       CONTINUE
C5    CONTINUE
C  ...EXIT FROM PROGRAM
      STOP
      END
```

Figure C. As proposed by Hull, comment cards have been inserted into the program shown in Fig. A, and statements have been indented in a systematic manner.

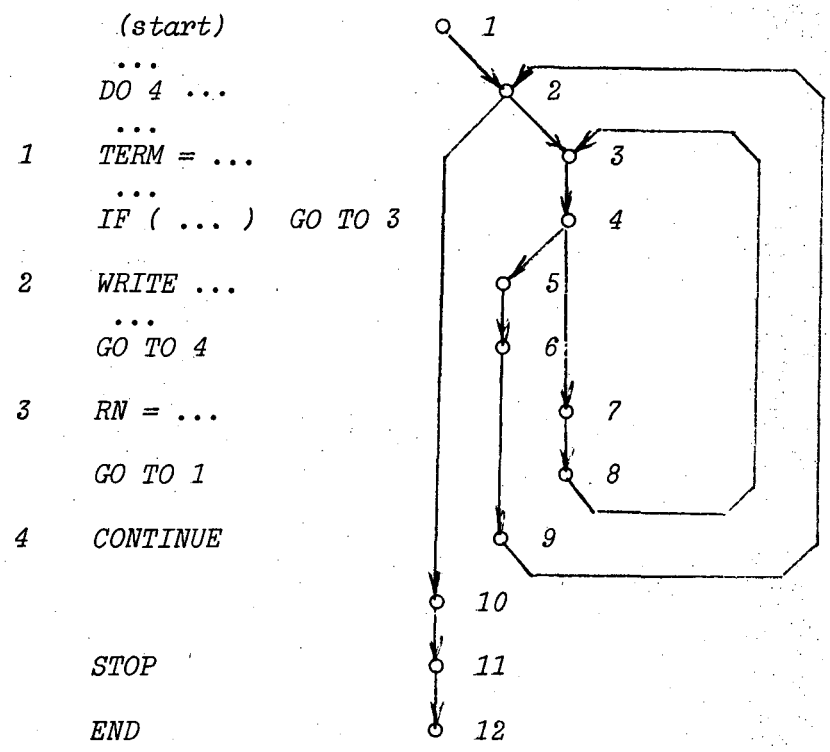


Fig. D. A flow graph corresponding to the program in Fig. A. The node numbers are consecutive, and do not necessarily agree with statement numbers.

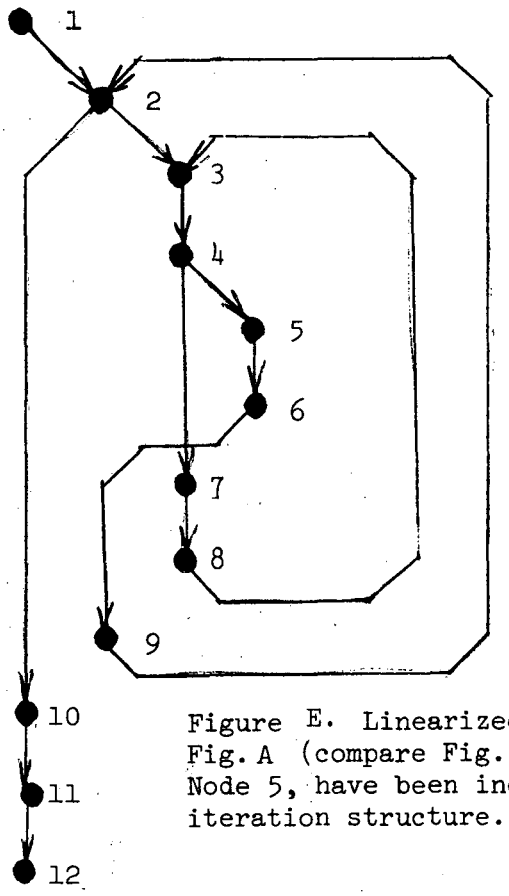


Figure E. Linearized flow graph of the program in Fig. A (compare Fig. D). The output statements, Node 5, have been included within the innermost iteration structure.

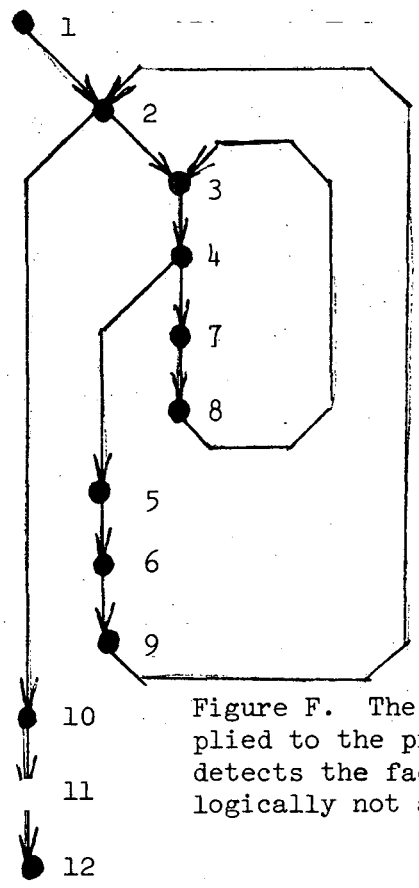


Figure F. The modified algorithm has been applied to the program of Fig. A. This version detects the fact that the output statements are logically not a part of the inner loop.

```
                *C      start
1 → (2)                 FUNCTION TER (X, Y, N, A, J)
                        DIMENSION X(N), Y(N)
                        NM1 = N - 1
                *C      L12:
                *C          L8:
2 → (8, 3)                  IF (A .LT. X(1)) GO TO 2
                *C          ...
                *C              L7:
                *C                  L5: begin iterative structure
3 → (7, 4)                          DO 1 I = 1, NM1
                *C                ......
4 → (6, 5)                          IF (A .GE. X(I + 1)) GO TO 1
                *C                ...
- - - - - - -  *C                  sequence break
6 → (3)↑          1                 CONTINUE
                *C                  return arc
- - - - - - -  *C                  sequence break
                *C                end L5
5 → (10)                          J = I + 1
                                  GO TO 3
- - - - - - -  *C                  sequence break
10 → (11)         3                 CONTINUE
                                    TER = Y(I) + ((A - X(I)
                              1         / (X(J) - X(I))) * (Y(J) - Y(I)))
                 C                  MANY MORE STATEMENTS MIGHT APPEAR HERE
11 → (12)                           RETURN
                *C              .........
- - - - - - -                      sequence break
                *C              end L7
7 → (8)
                *C          ...
                *C          end L8
8 → (9)           2         CONTINUE
                  C         ERROR...NO TABLE ENTRY FOUND
                            J = 0
                            TER = 0
9 → (12)                    RETURN
                *C          ...
- - - - - - -  *C              sequence break
                *C          end L12
12                          END
```

Figure G. Restructured listing of interpolation subprogram, based on the linearized flow graph of Fig. 4. A representation of the flow graph appears on the left. Comments preceded by an asterisk were generated by the analysis algorithm.

TECHNICAL INFORMATION DIVISION
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720