To improve the readability of programs over existing techniques, a new program representation termed GREENPRINT has been developed and is discussed in this paper. GREENPRINTs (the name taken from the phosphor fluorescence of certain display terminals and paralleling the term blueprints) are tree-structured diagrams together with source code statements that represent the control structure of programs. Discussed in this paper are the diagramming conventions, control flow methodology, presentation graphics, and practical experience with GREENPRINTs.

GREENPRINT: A graphic representation of structured programs

by L. A. Belady, C. J. Evangelisti, and L. R. Power

Flowcharts are the oldest graphic representations of programs. The works of Goldstine and von Neumann contain many flowcharts.[1] Largely due to processor speed and storage space limitations, early programs were not structured; branching to common code was important and occurred frequently. Later, high-level languages appeared and programs automatically generating flowcharts from program text were developed.[2] At the same time, program structures improved. Nassi-Shneiderman Diagrams (NSDs) were proposed much later to represent structured programs.[3] In this form, such program constructs as if-then-else and loop are represented as nested boxes. With a high level of nesting, these charts become wide, and their elements vary in size. HIPO charts attempt to capture the data flow of program segments by focusing on the representation of input data, process, and output data for program blocks.[4] Combinations of NSDs and HIPOs can be found in the literature,[5,6] and in some instances NSDs have been automatically generated.[7]

Further improvement can be achieved by direct input of charts using interactive graphics. The earliest general-purpose graphics system was Sketchpad.[8] More specialized approaches include block diagramming[9] and, more recently, the direct input of NSDs.[10,11] In the latter case, program text is automatically generated from NSDs. A recent example of the use of graphics in software design is the TELL system,[12] where NSDs are used for detailed program description.

The problem with the above graphics schemes is that source statements in a program listing, as the programmer "normally" views them, do not line up with their associated elements in the graphics representation. Thus, switching attention from one representation to the other can involve a lengthy search for the corresponding entity.

This paper discusses a research effort to study this problem and to try to devise an improved solution. The solution has been called GREENPRINT after the color of the CRT display. GREENPRINT diagrams, the result of the research effort and the subject of this paper, are aligned with formatted source code listings and can be printed side by side with them. Also, GREENPRINTs are suited to inexpensive devices, and can be used for program design or documentation.

GREENPRINTs in general

Just as an engineer studies a blueprint, a programmer may interpret two-dimensional green shapes (if the phosphor is such) at a CRT terminal. A GREENPRINT uses interconnected shapes to show the block structure and the control flow of a program. The detailed program text—the "bill of materials"—completes the part specification.

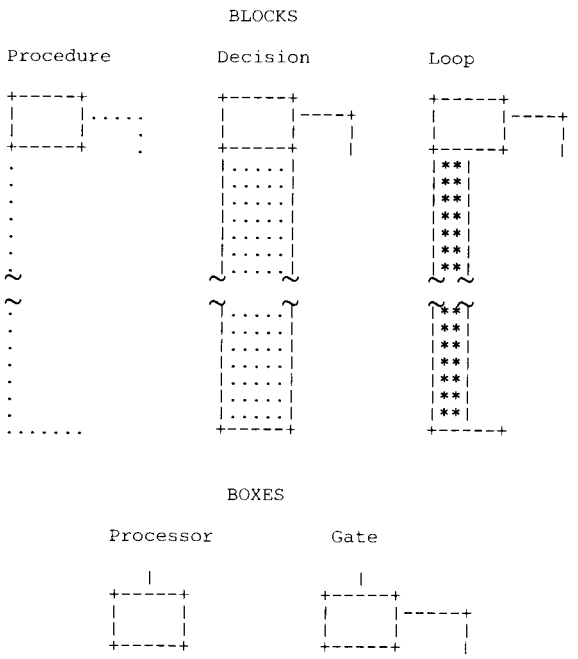
Many phases of the program development/maintenance process could use GREENPRINTs. During design, detail is suppressed, but an overview of the entire software system is given. Later, program logic is detailed in GREENPRINTs; then program text is written complementing the former. Finally, in maintenance, when more than ever the understanding of programs written by others is crucial, GREENPRINTs, the authors believe, can increase the productivity of program modification.

GREENPRINT was developed as a result of the authors' own difficulty, often frustration, in working with large programs written by others. The current version, which is described here, has evolved gradually. The authors have found it to be a useful tool.

GREENPRINT diagram

A GREENPRINT is a diagrammatic representation of a program drawn next to its program source listing. The diagram consists of only two types of *objects—blocks* and *boxes*. Blocks are used to illustrate program control statements and their scope (e.g., IF, DO WHILE); boxes are used to illustrate all other program statements. To represent a program, objects are connected and arranged over a virtual *grid* that outlines *rows* and *columns*. Rows correspond to

Figure 1 GREENPRINT objects



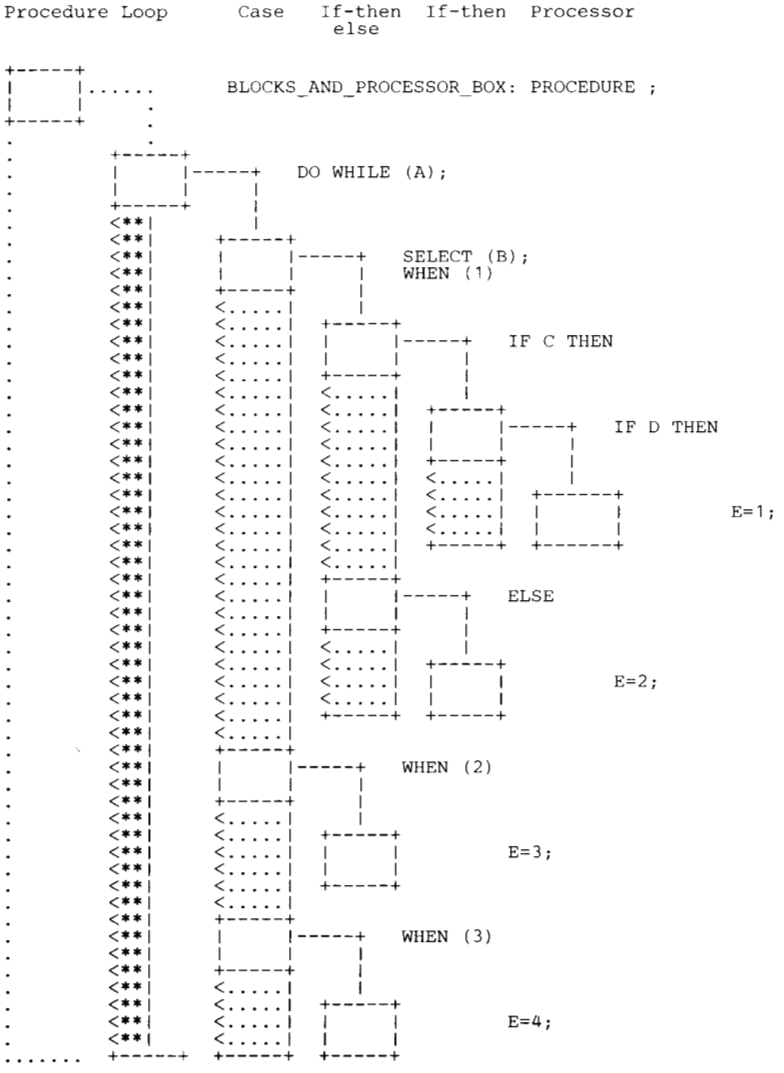
program statements or groups of statements; columns correspond to program block structure nesting.

Figure 1 shows a *procedure block*, a *decision block*, and a *loop block*. Each such block consists of a *pillar* and at least one *gate box* on the top of the pillar (the decision and procedure block may have additional gate boxes along the pillar). Each type of block has a different pillar to distinguish it visually. The figure also shows a *processor box*, distinguished from the *gate box* by the absence of any line to the right. A procedure block defines and spans the contents of a program or subroutine. Decision blocks represent if-then, if-then-else, and case statements. Loop blocks correspond to iterative DO-blocks. A gate box is always part of a procedure block, a decision block, and a loop block; a processor box stands alone. As examples in the paper show, a GREENPRINT representation of a program is a tree where blocks and processor boxes are nodes with the entry at the top and exits at the bottom or on the right. A gate box starts a subtree in the column immediately to its right. Figure 2 shows a GREENPRINT of a procedure with a loop, three types of decision blocks, and processor boxes. (The meaning of the "<"s on the left of pillars is discussed later.)

program text

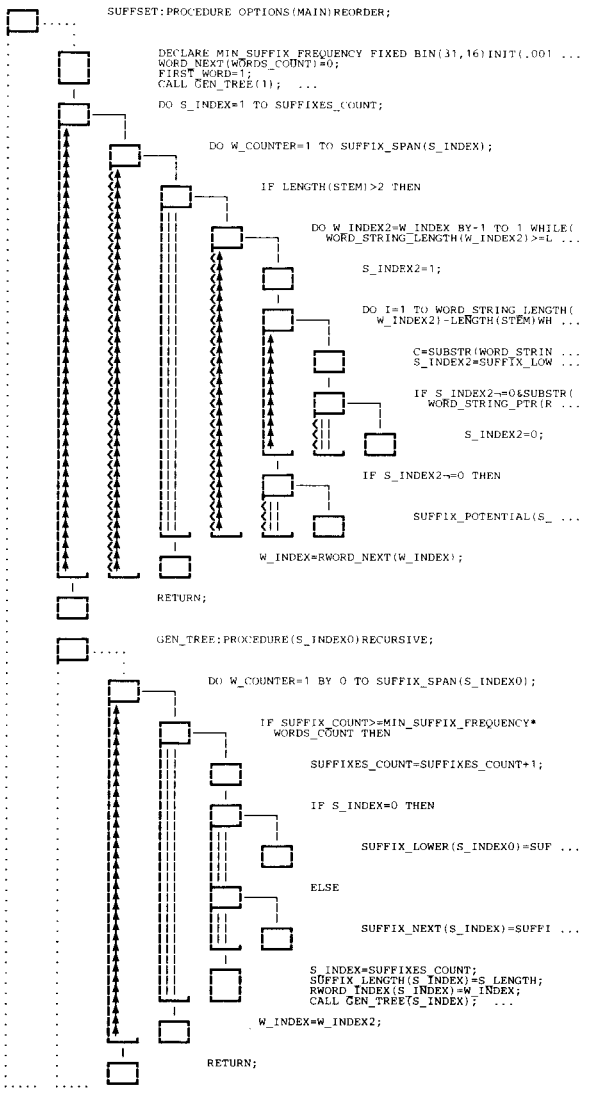
The processor box represents a segment of sequential statements (straight-line code), and a gate box refers to a predicate (condition) to control either a decision or a loop. The gate at the top of a

Figure 2 GREENPRINT diagram of a procedure block, loop block, three forms of decision block, and processor boxes



procedure block points to the body of the program. Corresponding text is written immediately to the right of a box. GREENPRINTs are arranged so that there is one box in each row, and the box is the rightmost object in that row. (Extra space may be introduced between rows to accommodate program text.) Figure 3 shows a GREENPRINT with associated text. It can be seen that the right contour of a GREENPRINT follows the indentation of the text and corresponds to the nesting of the program control structures. Since the GREENPRINT diagram is right next to, and in the same order as the program source text, the programmer can easily

Figure 3 A GREENPRINT with indented text



switch attention from diagram to text and back—a great time saver when studying programs. (The modified appearance of loop and decision pillars in Figure 3 is discussed later in this paper under the heading "Presentation Media.")

static block structure

Processor boxes interspersed with decision blocks and loop blocks correspond to the static block structure of a program. A left-to-right scan across a GREENPRINT reveals the decomposition of blocks and therefore the structure of the program. The procedure block is used to illustrate subroutine nesting. Procedure blocks always appear on the left, leaving the control flow of the

program displayed on the right. Figure 3 is a moderately complex example of a GREENPRINT. It shows a procedure in column one that consists of a procedure body in the top half of column two and a nested procedure in the bottom half of column two. The procedure body consists of a process box for initialization followed by a doubly nested loop (columns two and three) and a RETURN. The innards of the double loop consist of a decision (column four) to perform another loop (column five) whose body (column six) includes an inner loop and a decision. The decomposition of each block appears in the next column along a left-to-right scan. Notice also that all the processor boxes can be readily seen from the right without any obstruction. The gate boxes, also seen from the right, are partly obscured by right-angle legs exiting from the box. Thus boxes for sequential code and predicates (partly obscured) can be easily seen and discriminated by the user.

The programming language delimiters DO and END, essential for compiling the one-dimensional program text, become redundant because they are implied by the two-dimensional arrangement of blocks and boxes. If a GREENPRINT were used for actual coding, these delimiters could be automatically inserted before compilation for correct language syntax.

Dynamic program execution

Although a GREENPRINT is a tree, it does represent the flow of control of a structured program. While "playing machine" on a GREENPRINT (i.e., tracing control flow), the execution sequence generally progresses downward. Upon each entry to a block, at most one gate forces execution to continue in the next column to the right. The selection of a gate is determined by the truth value of the predicate for the box. An object with no successors beneath it is called a *terminal*. The pillar of a terminal decision block or loop block is tagged with <s on the left edge to facilitate tracing.

Flowcharts explicitly draw all flow of control lines. GREENPRINTs, which accentuate program block structure, omit the flow of control lines from terminal blocks and terminal processor boxes. Instead, the following rule is applied upon completing execution of a terminal object:

1. Move left (as suggested by the <s on terminal blocks) to the next loop block or nonterminal decision block, whichever comes first.
2. If it is a loop block, go up to its gate box to reevaluate its condition. If it is a nonterminal decision block, go down to the next sequential object in the same column.

With a little practice this rule becomes second nature and can be applied at a glance.

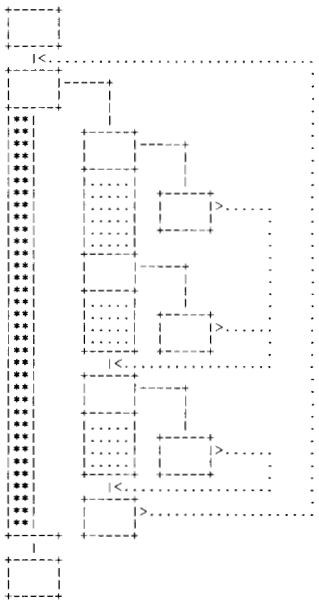
nonstructured flow of control

With the above rule, a GREENPRINT defines the flow of control for structured code. Nonstructured flow of control is indicated in a GREENPRINT by special processor boxes. Constructs such as GOTO, CALL, RETURN, and LEAVE are such examples. As opposed to a regular processor box containing possibly many sequential statements, the special version always represents a single statement, which is indicated in the associated text. By drawing these boxes differently (e.g., see the GOTO boxes in Figure 6, shown later) nonstandard flow of control can be highlighted. Auxiliary lines can be added to a GREENPRINT to show the flow of control for simple GOTO statements. This has not been done in the current work, which has concentrated on moderately well-structured code. If GOTO statements are relatively rare, merely highlighting them is adequate, and the diagram remains clean. Also note that the CALL statements in Figures 3 and 6 have not been highlighted by special processor boxes because their highlighting is considered optional.

GREENPRINTs and other charts

We have already shown how GREENPRINTs are related to indented text. Now we show that, as a program tree, GREENPRINT also spans a conventional flowchart. Observe the modified GREENPRINT in Figure 4. Note that it has auxiliary exit lines from the processor boxes drawn for the purpose of explanation. Clearly, the move-left-on-terminal rule previously described is equivalent to these lines. However, the resulting flowchart, thanks to the GREENPRINT drawing rules, highlights the program block structure. If the underlying program is GOTO-free, these rules contain the same information as the auxiliary lines and can therefore be omitted. Again, auxiliary lines can be added to GREENPRINTs to flag nonstructured program flow.

Figure 4 A GREENPRINT with auxiliary control lines added to form a flowchart



Further, Figure 5 shows transformations of both a flowchart and a Nassi-Shneiderman Diagram (NSD) into a GREENPRINT. The original charts, Figures 5A and 5E, show a loop around an if-then-else. Both transformed charts, Figures 5B and 5D, show blocks and boxes pushed to the right. The resulting GREENPRINT, Figure 5C, is shown with auxiliary control lines added.

Uses of GREENPRINT

There were two goals behind the GREENPRINT study. The first was to draw graphics images of existing program text. Indeed, the very need to understand complex and often obscure code written by others led the authors in the first place to develop GREENPRINT. An experimental program driven by a Backus-Naur Form grammar for PL/I was written to generate data for a drawing pro-

gram that produces a file to be displayed or printed. The best candidates for using these automatically generated GREENPRINTs are likely to be maintainers who must study and modify programs unfamiliar to them.

Secondly, GREENPRINT can be a design tool, a notation to first capture ideas as they emerge. Program design thus becomes drawing GREENPRINTs and entering predicates for gate boxes and sequential statements for process boxes. Since a GREENPRINT is precise, with text associated with each box, manual translation into conventional text is not required. Rather, an automatic transformation of GREENPRINTs and associated text into source statements can precede compilation. As a result, GREENPRINT can be the only program representation, also serving as documentation, whether it represents design or is generated from existing code. One of the authors designed the drawing program by using freehand GREENPRINTs.

GREENPRINT was originally developed specifically for use with IBM 3270 type devices, which are today widely available to programmers. Our current, batch-oriented implementation is used with these terminals and various types of printers. The experimental GREENPRINT drawing program has been parameterized so as to accept user-defined graphics elements corresponding to different source language constructs. This has encouraged user experimentation and led to the introduction of the stylized GOTO-box in Figure 6. Figures 3 and 6, printed on a photocomposer, were generated from the GREENPRINT drawing program by parametrically respecifying the GREENPRINT graphics elements, using an appropriate font. The up-arrow in the loop pillar enhances tracing the flow of control in Figure 3.

An interactive GREENPRINT, which has not been studied, would require only a few commands to support the placing of blocks and boxes at points on a grid. The machine could facilitate this process in several ways. For example, the most recently placed object is terminal by default but changes automatically to nonterminal when a new block or box is suspended from it. The system refuses to accept a second box in the same row, such as a processor box immediately following a processor box (except for special processor boxes) or a stand-alone gate box. Also, as a subtree grows downward, so do all pillars of the enclosing blocks to the left of the subtree, automatically.

To teach programming to a novice, to train programmers, to stimulate insight of designers, or to facilitate the exploration of alternative designs, media other than printers and display terminals come to mind. Imagine, for instance, prefabricated and possibly colored magnetic blocks and boxes placed on a metal board with a marked grid. Programming or its demonstration could then be-

Figure 5 Transformations from a flowchart and an NSD to a GREENPRINT: (A) Flowchart; (B) Transformed flowchart; (C) GREENPRINT; (D) Transformed NSD; (E) NSD

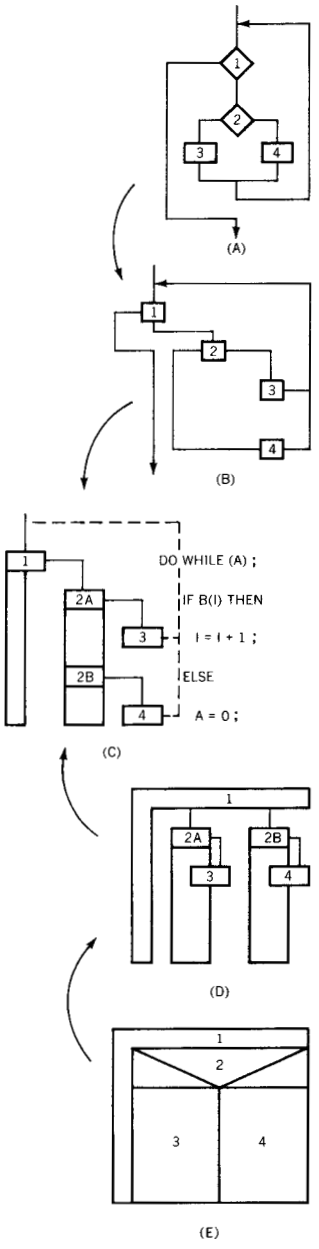
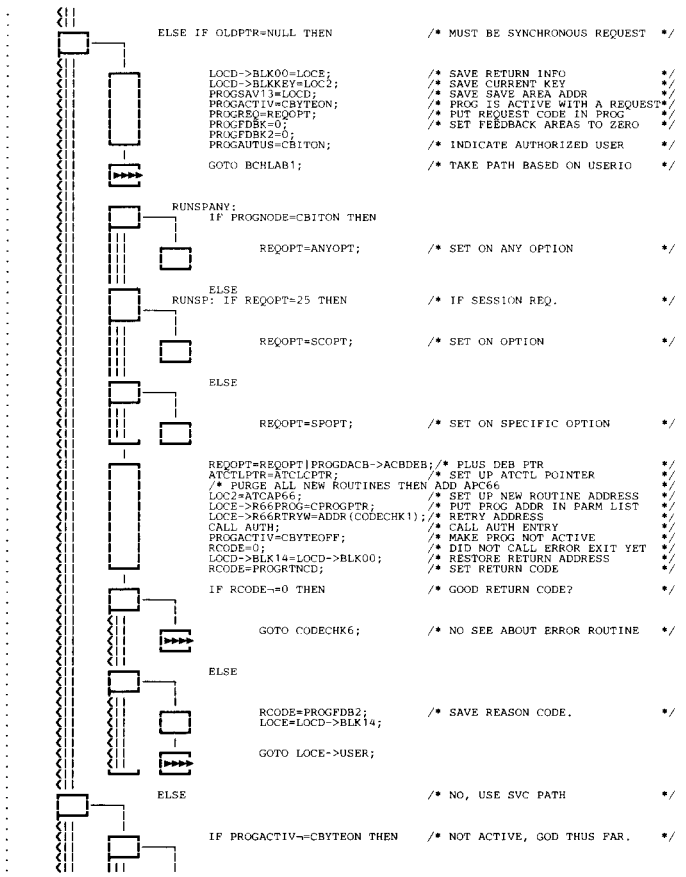


Figure 6 GREENPRINT of a poorly structured program



come the exploration of different arrangements of processor boxes, loops, and decisions into trees. This would be similar to a breadboard used in electrical engineering.

GREENPRINT practical experience

In practice, large, complex programs have benefited most from the use of GREENPRINTs. Understanding the flow of control is one of the first steps in understanding a complex program. This can be accomplished by studying a hard copy listing of the GREENPRINT diagram printed beside the source program. The side-by-side, linear correspondence of the GREENPRINT diagram to the source code is a very useful and unique feature of GREENPRINTs. Since the GREENPRINT diagram can be produced directly from a parse of the source code, it would be useful and appropriate for a compiler to optionally append the GREENPRINT diagram to the output listing.

It has been discovered that GREENPRINTs produced from poorly structured source code are of special value. Although GREENPRINTs take maximal advantage of the block structuring expressed in source code control structures, they are not restricted to them. Indeed, the desired benefits of block structuring can easily be subverted by a few GOTO statements. Figure 6 is an example of this, being a portion of an actual systems program recognized as a maintenance problem. The GREENPRINT diagram highlights a poorly structured sequence of code that the neatly indented source code hides. Notice the following about the sequence starting at the label RUNSPANY: (a) it can be reached only by a GOTO; (b) it consists of three blocks—a decision block, followed by a processor box, followed by a decision block; and (c) the only way to exit this sequence is via one of the two GOTO statements in the third block. Consequently, this sequence, although nominally embedded within one leg of a decision block, could be moved elsewhere without affecting the logic of the program, thus improving the structure of the code. This flaw in the code was discovered in a few moments by inspecting the GREENPRINT. Examination of a traditional, automatically generated flowchart of this same program did not reveal this flaw. The indented source code masked the flaw, and, because it is poorly structured, the program is not expressible as an NSD.

A side issue of user experience concerns source program comments. Although comments are usually a valuable form of program documentation, they often do not describe a program's flow of control. They may instead document data structures, or describe the intent of a program at a more abstract level. Consequently, some users have observed that suppressing comments in a GREENPRINT clarifies the flow of control of a program by eliminating nonessential information primarily concerned with other aspects of the program.

complexity

We propose the following research topic: Deduce certain programming measures from size and shape characteristics of GREENPRINTs. For example, the jaggedness of the right contour could be used to characterize or classify programs with respect to structure, style, or complexity.

A hypothesis could be studied that the average width of a GREENPRINT is proportional to the expected reading rate of a programmer or the comprehension complexity of a program based on the following expression:

$$Average\ width = \frac{1}{r_{total}} \sum_{r=1}^{r_{total}} K_r$$

where K_r is the number of occupied columns in row r and r_{total} is the number of rows in the GREENPRINT; and the total complexity C is given by the following equation:

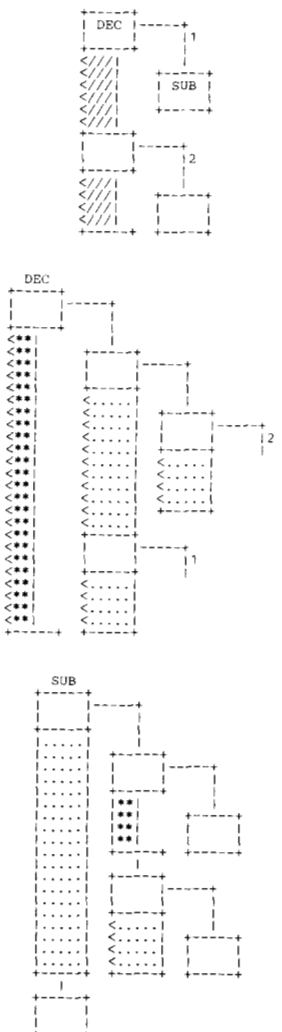
$$C = \frac{S}{r_{total}} \sum_{r=1}^{r_{total}} K_r$$

where S is the total number of program statements. This complexity measure accounts for both the average nesting and the total length of a program. Since the processor box count does not contain the number of sequential statements in the program, the length of the program S is used.

GREENPRINTs as overviews

Sometimes an overview of a large program is required at the expense of detail. Two methods are envisioned for this. In the first, which has not been studied in depth, a box or block may stand for an undetailed program segment of any size, and it may contain the name of the segment. In this manner, a subtree can be replaced by a named processor box. Such a facility is important while designing in a top-down fashion. Also, a block, similar to a decision block, can represent a program segment that determines which gate is to receive control. Such a block can be more general in the sense that it gives control to different gates, depending on an algorithm. Figure 7 shows a summary GREENPRINT at the top and detailed GREENPRINTs SUB and DEC below. SUB (for subroutine) illustrates a detailed GREENPRINT and DEC (for decision) represents a program that plays the role of a case-statement predicate. DEC transfers control at exits 1 or 2, thus—at both levels—representing actions to be performed. The pillar of the high-level block is altered to indicate that it is not a standard block. Extending this notion, GREENPRINTs can be used to represent any tree-structured information. By the appropriate design of pillars, boxes, and connectors, the entities and their relations can be depicted graphically.

Figure 7 A high-level GREENPRINT with two detailed GREENPRINTS



In the second method for overview, as exemplified in Figures 3 and 6, all blocks and boxes can be shrunk horizontally and vertically, even to a single character, thus allowing the display of the control flow of a large program in a smaller area. Figures 3 and 6 were automatically generated and then printed with an appropriately small print font. In addition, some of the program source text was elided in Figure 3.

Concluding remarks

GREENPRINT as a graphics representation of program control structure is unique in that its objects—blocks and boxes—appear from top to bottom in the same order as the associated program text. The two representations can thus be studied and worked with concurrently. Other advantages, some shared by conventional flowcharts and NSDs, include the capability of automatically generating GREENPRINTs from program text and generating control statements from a GREENPRINT. GREENPRINTs can be dis-

played on inexpensive terminals. In addition, the exactness of GREENPRINTs suggests the possibility of developing program complexity metrics based on purely geometric properties. But this and the extension of the GREENPRINT approach to include structure and flow of data remain interesting research topics at this time.

ACKNOWLEDGMENTS

The authors wish to thank J. Cavanagh for nourishing GREENPRINT when it was a seedling, and H. Ellozy for actively using the program and discovering uses in analyzing poorly structured parts of programs.

CITED REFERENCES

1. H. H. Goldstine and J. von Neumann, "Planning and coding problems for an electronic computing instrument," *John von Neumann, Collected Works, Volume V*, A. H. Taub (General Editor), The Macmillan Company, New York (1963), pp. 80-235.
2. L. M. Haibt, "A program to draw multilevel flow charts," *Proceedings of the Western Joint Computer Conference, The Joint IRE-AIEE-ACM Computer Conference*, San Francisco, CA, March 3-5, 1959, published by the Institute of Radio Engineers (now IEEE), New York (1959), pp. 131-137.
3. I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices* **8**, No. 8, 12-26 (August 1973).
4. J. F. Stay, "HIPO and integrated program design," *IBM Systems Journal* **15**, No. 2, 143-154 (1976).
5. N. Chapin, "New format for flowcharts," *Software—Practice and Experience* **4**, No. 4, 341-357 (October-December 1974).
6. K. T. Orr, *Structured Systems Development*, Yourdon, Inc., New York (1977).
7. P. Roy and R. St-Denis, "Linear flowchart generator for a structured language," *ACM SIGPLAN Notices* **11**, No. 11, 58-64 (November 1976).
8. I. E. Sutherland, "Sketchpad, a man-machine graphical communication system," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **23**, 329-346 (1963).
9. L. A. Belady, M. W. Blasgen, C. J. Evangelisti, and R. D. Tennison, "A computer graphics system for block diagram problems," *IBM Systems Journal* **10**, No. 2, 143-161 (1971).
10. N. Ng, *A Graphical Editor for Programming Using Structured Programming*, Research Report RJ2344, IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193 (1978).
11. R. Williams and G. M. Giddings, "A picture-building system," *IEEE Transactions on Software Engineering* **SE-2**, No. 1, 62-66 (March 1976).
12. P. G. Hebalkar and S. N. Zilles, *TELL: A System for Graphically Representing Software Designs*, Research Report RJ2351, IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193 (1978).

The authors are located at the IBM Thomas J. Watson Research Center, P.O. Box 218, Route 134, Yorktown Heights, NY 10598.

Reprint Order No. G321-5137.