First-year results from a research program on human factors in software engineering

by SYLVIA B. SHEPPARD, BILL CURTIS, PHIL MILLIMAN, M. A. BORST, and TOM LOVE

General Electric Company
Arlington, Virginia

INTRODUCTION

For the past two years the Software Management Research Unit at General Electric has been investigating several areas of human factors in software engineering with support from Engineering Psychology Programs of the Office of Naval Research. There have been two major thrusts in this research. The first thrust investigated the effects of several modern programming practices on programmer efficiency. The second thrust investigated the prediction of programmer performance from software complexity metrics such as those proposed by Halstead and McCabe. This research program consisted of separate experiments on the understanding, modification, debugging, and construction of software, each using professional programmers. Each experiment investigated both the effects of experimentally manipulated programming practices, and the values of complexity metrics computed from the programs employed.

Structured coding techniques, mnemonic variable names and commenting are programming practices which supposedly reduce the complexity of software. Dijkstra contended that program construction should proceed in a structured, top-down fashion. By limiting the control structures allowed, he assumed that the simplified control flow would make functions performed by the program easier to trace. Mnemonic variable names supposedly simplify the cognitive task of understanding a program by reducing the memory load on a programmer. The inclusion of comments purportedly simplifies modification tasks, although there are different methods of commenting. Global comments preceding a program summarize what objectives are accomplished, while in-line comments delineate how and where the objectives are fulfilled.

In 1972 Halstead first published his theory of software physics (renamed software science) stating that algorithms have measurable characteristics analogous to physical laws. These characteristics provide one assessment of program complexity. According to Halstead, the amount of mental effort required to generate a program can be calculated from simple counts of distinct operators and operands and the total frequencies of operators and operands. From these four quantities Halstead derives the number of mental comparisons required to generate a program. Correlations often greater than .90 have been reported between Halstead's metrics and such dependent measures as the number of bugs in a program, programming time, and the quality of programs.

More recently, McCabe developed a definition of complexity based on the decision structure of a program. McCabe's complexity metric is the classical graph-theory cyclomatic number which represents the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. Simply stated, McCabe counts the number of elementary control path segments. When combined these segments generate every possible path through the program.

This paper reports results from the experiments on understanding and modification conducted during the first year of this research program. The first experiment investigated the effect of structured coding and mnemonic variable names on program comprehensibility. The second experiment studied the effects of structured coding and global versus in-line comments on modification tasks.

METHOD

Participants

In each experiment 36 programmers were tested in several General Electric locations. Participants in Experiment 1 had working knowledge of FORTRAN and averaged 6.8 years of professional programming experience (SD = 5.8). In Experiment 2 the participants had an average of 5.9 years of professional programming experience (SD = 4.0), a working knowledge of FORTRAN, and none had participated in the previous experiment. The majority of participants came from an engineering background.

Procedure

In both experiments a packet of materials was prepared for each participant with written instructions on the experimental tasks. As a preliminary exercise, all participants were presented the same short FORTRAN program and a
brief description of its purpose. In Experiment 1 they studied this program for 10 minutes and were then given 15 minutes to reconstruct a functionally equivalent program from memory. In Experiment 2 all participants were asked to modify the same short FORTRAN program. They were given a brief description of its purpose and were allowed unlimited time to complete a specified modification. This introductory program provided a common basis for comparing the skills of participants and diminished learning effects prior to the experimental tasks. This latter point is important since a pilot study indicated that learning may occur during this type of task.

Following the initial exercise, participants were presented in turn with three separate programs comprising their experimental tasks. In Experiment 1 they were allowed 25 minutes to study each program, during which they were permitted to make notes or draw flowcharts. At the end of the study period, the original program and all scrap paper were collected. Each participant was then given 20 minutes to reconstruct a functional equivalent of the program from memory on a blank sheet of paper, but was not required to reproduce the comment section. In Experiment 2 one modification was requested for each of the three programs and was described on a sheet accompanying the program listing. Participants worked at their own pace, taking as much time as needed to implement the modification. A break of 15 minutes occurred before the last program was presented in each experiment.

Independent variables

Program class. Three general classes of programs were used in Experiment 1: engineering, statistical, and non-numerical. Three programs were employed from each class with lengths varying from 36 to 57 statements. These nine programs were selected from among many solicited from programmers at several locations and were considered representative of programs actually encountered by practicing programmers. All experimental programs were compiled and executed using appropriate test data. Experiment 2 used three of the nine programs from Experiment 1.

Complexity of control flow. Three control flow structures were defined for each program in both experiments. Structured control flow was generally consistent with the tenets of structured programming described by Dijkstra. When the rules for structured programming are applied rigorously, awkward constructions may occur in standard FORTRAN such as DO loops with dummy indices. In a second version of each program, these awkward constructions were largely eliminated with a more naturally structured control flow. These conventions included multiple returns, exits from DO loops, and judiciously used backward GO TO's. In the unstructured version of each program, the control flow was not straightforward. Expanded DO loops, arithmetic IF's, and unrestricted use of GO TO's were allowed.

Variable name mnemonicity. In Experiment 1 three levels of mnemonicity for variable names were developed. The programs were shown to several non-participants who were asked to assign names to the variables. The names chosen most frequently were used in the most mnemonic condition. The medium mnemonic level consisted of less frequently chosen names. In the least mnemonic condition, names consisted of one or two randomly chosen alphanumeric characters.

Comments. Three levels of commenting were tested in Experiment 2: global, in-line, and none. Global comments provided an overview of the function of the program and identified the primary variables. In-line comments were interspersed throughout the program and described the specific functions of small sections of code.

Modifications. Three types of modifications were selected for each program in Experiment 2 as typical changes a programmer might be expected to implement. The level of difficulty for seven of the nine modifications increased as more lines had to be added to the original code, and the hardest modifications for each program required the most additional lines.

Experimental design. In order to control for individual differences in performance, a within-subjects factorial design was employed in each experiment. In Experiment 1 three types of control flow were defined for each of nine programs, and each of these 27 versions was presented in three levels of variable mnemonicity, for a total of 81 programs. In Experiment 2 three levels of control flow were defined for each of the three programs. Each of these nine versions was presented with one of three levels of documentation. Modifications at three levels of difficulty were developed for each program, generating a total of 81 experimental conditions. The first 27 participants in each experiment exhausted the total of 81 programs. The additional nine participants repeated 27 of the previous experimental tasks. Programmers at each location were randomly assigned to experimental conditions in order that they would experience each level of each independent variable. That is, within their three tasks they worked with a program from each class, with each type of control flow, and at each level of documentation (variable mnemonicity or type of commenting). Each of the first 27 participants experienced unique combinations of these levels across their three experimental tasks. The order of presentation of the three programs was assigned randomly to each participant.

Covariates. In order to obtain a measure of programming ability related to the experimental tasks, scores on the preliminary tasks in both experiments were used as a covariate. Participants reported their type of programming experience and the number of years they had been programming professionally. Order of presentation was a situational covariate.

Complexity measures

Halstead's E. Halstead's effort metric (E) was computed precisely from a program (based on Reference 22) whose input was the source code listings of the 27 distinct programs in each experiment. Programs differing only in variable mne-
monocity or type of commenting were not considered distinct programs in this analysis. The computation formula was:

\[ E = \frac{n_1 N_2 (N_1 + N_2)}{2 n_1} \log_2 \left( \frac{n_1 + n_2}{n_1} \right) \]

where

- \( n_1 \) = number of unique operators
- \( n_2 \) = number of unique operands
- \( N_1 \) = total frequency of operators
- \( N_2 \) = total frequency of operands

**McCabe's \( v(G) \).** McCabe's metric is the classical graph-theory cyclomatic number defined as:

\[ v(G) = \text{# edges} - \text{# nodes} + 2 \text{ (connected components)} \]

McCabe presents two simpler methods of calculating \( v(G) \). His metric equals the number of predicate nodes plus 1. Values of \( v(G) \) can also be computed from a planar graph of the control flow by counting the number of regions.

**Length.** The length of the program was computed as the total number of FORTRAN statements excluding comments.

**Dependent variables**

**Experiment 1.** The criterion for scoring the programs in Experiment 1 was the functional correctness of each separately reconstructed statement. Variable names and statement numbers which differed from those in the original program were counted as correct when used consistently. Control structures could be different from the original program so long as the statements performed the same function. The score on each experimental task was the percent of statements correctly recalled. Three judges scored each program independently. Interjudge correlations of .96, .96, and .94 were obtained across the three sets of scores. The average of the three scores (percents of statements correctly reconstructed) for each program was the dependent variable in the data analysis for Experiment 1.

**Experiment 2.** The dependent variables for Experiment 2 were the correctness of the modification and the time taken by the participant to perform the task. The individual steps necessary for correct implementation of the requested modifications had been delineated in advance and assigned equal weights. That is, prototypes of each program with each modification correctly implemented were established as the criteria against which participants' work would be compared. A percentage score reflecting the correctness of each modification was computed by comparing participants' changes with the criteria. The time to write a modification was measured to the nearest minute by an electronic timer.

**Analysis**

Results were analyzed in two phases. The first phase investigated the effects of experimentally manipulated variables, while the second phase evaluated the performance predictions of the software complexity metrics. The experimental effects of programming practices were analyzed in hierarchical regression analyses. In these analyses domains of variables were entered sequentially into a multiple regression equation to determine if each successive domain added significant prediction to that afforded by domains already entered. Effects related to pre-existing differences among participants and programs were entered into analyses prior to evaluating the effects of programming practices. The variables representing the different conditions of experimentally manipulated variables were effect coded. Analyses investigating relationships among Halstead's \( E \), McCabe's \( v(G) \), number of statements, and performance were conducted with Pearson product-moment correlation coefficients.

**RESULTS**

**Experimental manipulations**

**Experiment 1.** Table I presents the results of the hierarchical regression analyses for Experiment 1. Figures presented in this and succeeding tables indicate the unique percent of variance contributed to the prediction of performance by a variable domain when added into the analysis with preceding domains. Significance levels identified by asterisks indicate the likelihood (expressed as a proportion) that a prediction of this significance could have occurred by chance.

An average of 50 percent of the statements were correctly recalled across all programs and experimental conditions. Pretest scores accounted for 17 percent of the variance among scores on the percent of statements correctly recalled. No relationships were observed for type and length of programming experience or job location.

Differences among the program classes accounted for 8 percent of the variance in performance scores in addition to that accounted for by individual differences among participants. Engineering programs were the most difficult (41 percent of the statements correctly recalled), followed by statistical (52 percent), and non-numeric (57 percent) programs. When the specific program was taken into account, an additional 20 percent of the variance in performance was explained. However, this result was not strictly a function of differences among programs, because variance related to

<table>
<thead>
<tr>
<th>Variable domain</th>
<th>( \Delta R^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pretest</td>
<td>.17**</td>
</tr>
<tr>
<td>Class of program</td>
<td>.08**</td>
</tr>
<tr>
<td>Specific program</td>
<td>.20**</td>
</tr>
<tr>
<td>Control flow complexity</td>
<td>.07**</td>
</tr>
<tr>
<td>Variable mnemonicity</td>
<td>.01</td>
</tr>
<tr>
<td>Total ( R^2 )</td>
<td>.53***</td>
</tr>
</tbody>
</table>

Note: \( n=108 \). **p \( \leq .01 \). ***p \( \leq .001 \).
specific programs was confounded with variance related to participants. That is, each participant saw only three of the nine programs. Overall, 45 percent of the variance in performance was accounted for by differences among participants and programs.

The complexity of the control flow affected performance, accounting for 7% of the variance in addition to that accounted for by differences among programs and participants. As expected, unstructured programs were the most difficult to reconstruct. A post hoc analysis showed the means for naturally structured and unstructured programs (56 percent versus 42 percent, respectively) to be significantly different (p ≤ .05). Performance on structured programs fell between these two. No differences occurred among levels of variable mnemonicity.

**Experiment 2.** Across all experimental conditions, an average of 62 percent of the steps for each modification were accurately implemented. The 108 accuracy scores ranged from five scores of 0 percent to 24 scores of 100 percent and were negatively skewed. The average time to complete the modifications was 17.9 minutes, ranging from 2 to 59 minutes with a positive skew. Accuracy and time were uncorrelated.

Table II presents hierarchical regression results for the accuracy of the participants' modifications. Only 19 percent of the variance in accuracy scores could be predicted by the variable domains studied. However, there were substantial differences in the degree to which performance on each of the three programs could be predicted. On two of the programs, 35 percent of the variance in accuracy scores was accounted for, while results for the third program were insignificant.

Order of presentation accounted for 5 percent of the variance in accuracy scores. Participants made more complete modifications in less time with each succeeding experimental task. However, the two programs on which performance proved most predictable were more frequently presented second or third in order. Thus, random assignment of presentation orders failed to counter-balance the number of times each condition appeared in each position order.

The difficulty of the modification accounted for 9 percent of the variance in accuracy scores on the two most predictable programs. Performance was poorer on modifications which required more lines of code to be inserted. The complexity of the control flow accounted for 7 percent of the variance in accuracy scores on the two programs for which accuracy was most predictable. Modifications made to structured programs were more accurate than those made to unstructured programs. Accuracy scores did not differ among programs, nor among the type of comments included in the program.

Table III presents hierarchical regressions for time to completion. Across all three programs, 28 percent of the variance in the time required to complete the modifications could be accounted for by variables studied here. Time to complete the modifications was more easily predicted than accuracy scores across all three programs.

Results of the hierarchical regression for time were generally similar to the results observed for accuracy. The specific program and type of comments were unrelated to the criterion. Significant variance was accounted for by both the difficulty of the modification and the order of presentation. Again, however, the interpretation of the effect for this latter variable is confounded. Neither the pretest scores nor control flow complexity were significantly related to time, although they had been modestly related to accuracy.

Further inspection verified that the number of additional statements required in the code to accurately complete a modification was related to the time required to insert them. Fitting a curvilinear function to these data using least squares procedures resulted in a curvilinear correlation (second order polynomial) of .80 (p ≤ .05) and a standard error of 2.53 minutes. No such relationship was found for accuracy.

**Software complexity measures**

Since different levels of variable mnemonicity and type of commenting neither affected performance, nor caused any change in the value of the complexity metrics for a particular program, the data reported in this section were aggregated over the three levels of mnemonicity in Experiment 1 and type of commenting in Experiment 2. This procedure resulted in 27 data points for each experiment. Each datum represented the average of at least three performance scores. Table IV presents the correlations among the three complexity measures in both experiments. Correlations in the lower triangle are from Experiment 1; those in the upper triangle are from Experiment 2. Generally these correlations were quite large in both experiments.

Table V presents correlations between the complexity
measures and performance criteria in both Experiments 1 and 2. In Experiment 1 the correlations between performance and each of the complexity measures were all negative, indicating that fewer lines were recalled as the level of complexity represented by these three metrics increased. Performance was moderately related to length and McCabe’s \( v(G) \), but not to Halstead’s \( E \).

Most of the significant correlations with performance in Experiment 2 were observed for measures computed on correctly modified rather than unmodified programs. Correlations reported in Table V were for measures computed on modified programs. All three measures were moderately correlated with time to complete the modification, while only length and McCabe’s \( v(G) \) were significantly related to accuracy.

The complexity of the control flow moderated the relationships between performance and the complexity metrics in both experiments. That is, while insignificant correlations were observed when the control flow was structured or naturally structured, this was not the case for unstructured code. Correlations with percent recalled correctly in Experiment 1 of \(-.55 (p \leq .001)\) and \(-.45 (p \leq .01)\) for \( v(G) \) and \( E \) were observed on unstructured programs. In Experiment 2 correlations relating time with Halstead’s \( E \) went from .08 in the structured code, to .28 \((p \leq .05)\) in naturally structured code, to .38 \((p \leq .05)\) in unstructured code. No such moderating effects were observed for McCabe’s \( v(G) \), nor for either metric with accuracy scores.

Correlations between the complexity metrics and performance criteria in Experiment 2 were also moderated by the type of commenting. When no comments were included in the program, significant correlations on modified programs for both Halstead’s \( E \) and McCabe’s \( v(G) \) were observed for both accuracy \((r = -.34 \text{ and } -.35, p \leq .05)\) and time \((r = .47 \text{ and } .44, p \leq .01)\). Insignificant correlations were usually observed when either global or in-line comments appeared in the code.

The amount of professional programming experience profoundly affected the relationships observed between the complexity measures and percent of statements correctly recalled in Experiment 1 and time to completion in Experiment 2. For programmers with three or less years of professional experience in Experiment 1, correlations of \(-.47 (p \leq .001)\) for McCabe’s \( v(G) \) and \(-.35 (p \leq .05)\) for Halstead’s \( E \) were observed. Insignificant correlations were observed for programmers with more than three years experience. For time to completion in Experiment 2, correlations of \(.55 (p \leq .001)\) for Halstead’s \( E \) and \(.52 (p \leq .001)\) for McCabe’s \( v(G) \) were observed for programmers with three or less years of professional experience, while no correlations above .20 were observed for programmers with more than three years experience.

**DISCUSSION**

**Experimental manipulations**

Several factors were consistently related to programmer performance. Individual differences among participants and the complexity of the control flow were found to influence programmer performance in both experiments. In Experiment 2 the difficulty of the requested modification and the order of presentation influenced both the accuracy and speed of implementing modifications. Each of these factors contributed independently to predicting program comprehension. Contrary to expectations, however, mnemonic variable names and types of commenting did not influence performance.

Control flow complexity was significantly related to both the percent of statements correctly recalled in Experiment 1 and the accuracy of the modifications on two of the programs studied in Experiment 2, but not to the time spent implementing modifications. In Experiment 1 naturally structured code was more easily comprehended than unstructured code. In Experiment 2 more accurate modifications were made to structured rather than unstructured code. It is not clear from the results of these two experiments whether rigidly structured code or code structured with a more natural control flow for FORTRAN can be maintained more efficiently. However, both of these control flows proved superior to unstructured code in at least one of the experiments.

Differences among programs played an important, but difficult-to-explain, role in these experiments. Effects on performance attributed to these differences may have resulted from some familiarity factor specific to the samples of programs and programmers studied. Further, effects due to differences among specific programs were confounded in Experiment 1 with effects related to individual differences among participants.

It is not surprising that the difficulty of a modification in Experiment 2 was related to the time required to implement it. The significant factor in the time spent implementing a modification was the number of new lines to be added rather than the number of in-line changes, such as deletions or substitutions. The difficulty of a modification also affected the accuracy with which it was implemented. Greater cog-

---

**TABLE IV.**—Intercorrelations among Complexity Measures

<table>
<thead>
<tr>
<th>Complexity Measure</th>
<th>( E )</th>
<th>( v(G) )</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halstead’s ( E )</td>
<td>.88***</td>
<td>.92**</td>
<td></td>
</tr>
<tr>
<td>McCabe’s ( v(G) )</td>
<td>.84***</td>
<td>.89***</td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td>.47**</td>
<td>.64***</td>
<td></td>
</tr>
</tbody>
</table>

*Note: \( n = 27 \). **\( p \leq .01 \). ***\( p \leq .001 \).*

**TABLE V.**—Correlations between Complexity and Performance Measures

<table>
<thead>
<tr>
<th>Complexity metric</th>
<th>Percent Recalled (Exp. 1)</th>
<th>Accuracy (Exp. 2)</th>
<th>Time (Exp. 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halstead’s ( E )</td>
<td>-.13</td>
<td>-.29</td>
<td>.44**</td>
</tr>
<tr>
<td>McCabe’s ( v(G) )</td>
<td>-.35*</td>
<td>-.36*</td>
<td>.38*</td>
</tr>
<tr>
<td>Length</td>
<td>-.53**</td>
<td>-.34*</td>
<td>.46**</td>
</tr>
</tbody>
</table>

*Note: \( n = 27 \). *\( p \leq .05 \). **\( p \leq .01 \).*
nitive difficulty appeared to be involved in creating new code than in merely deleting or altering it.

The inclusion of mnemonic variable names and either global or in-line comments were expected to improve programmer performance. The surprising lack of effects for documentation aids in both experiments may have occurred for several reasons. First, in Experiment 1 variable mnemonic was manipulated and global comments were provided with all programs. In Experiment 2 type of commenting was manipulated and mnemonic variable names were provided in all programs. Thus, the existence of one type of documentation may have reduced the additional information available from the documentation aid being experimentally manipulated, reducing its impact on performance.

A second possibility is that documentation aids do not contribute significantly to performance for programs of the modular size (35-55 lines) employed here. In large systems with many modules and thousands of lines of code, documentation may have more impact on performance because of the increased amount of information programmers must remember. Thus, program size may moderate the relationship between documentation and performance.

Finally, although mnemonic variable names did not affect performance in Experiment 1, many participants seemed to prefer them. That is, they used their own, more meaningful names when reconstructing the least mnemonic versions of the programs. For the medium and most mnemonic versions, they tended to use the original names supplied. Thus, the contribution of mnemonic variable names is supported by anecdotal rather than statistical evidence.

Results for the modern programming practices studied here were probably conservative due to the small size of programs studied. The cognitive load placed on programmers attempting to understand or modify approximately 50-line programs did not require the amount of assistance provided cumulatively by structured coding, mnemonic variable names, and comments. While the information provided by these practices was not necessarily redundant, the task could be mastered with less information than presented. In a larger system composed of many modules, however, the cognitive burden of implementing modifications may be so great that each of these programming practices may contribute significantly to efficiency. Thus, further research needs to assess the independent benefits of these practices in substantially larger programs.

Software complexity metrics

The two experiments comprising this study produced empirical evidence that software complexity metrics were related to the difficulty programmers experienced in understanding and modifying programs. Deeper analysis indicated, however, that the Halstead and McCabe metrics predicted programmer performance only on certain programs. Programs on which significant prediction was observed were characterized by the absence of programming practices such as structured coding or commenting which provided assistance in understanding the code. These complexity metrics were more predictive of the performance of less experienced programmers. A more complete presentation and discussion of these results is presented by Curtis, Sheppard, Milliman, Borst, and Love. 8

Assessment of the psychological complexity of software appears to require more than a simple count of operators and operands or basic control paths. Many programs have characteristics unassessed by these metrics which may heavily influence psychological complexity. For instance, the use of structured coding techniques or comments may reduce the cognitive load on a programmer in ways unassessed by the complexity metrics. Further, complexity metrics may not be capturing the most important factors for predicting the performance of experienced programmers who may either be conceptualizing programs at a level other than that of operators, operands, and basic control paths, or who can fit the program into a schema similar to one with which they have had previous experience.

Even though moderating effects were observed in these data, stronger relationships with performance may have been masked by the effects of differences between individuals and programs which were enhanced by limitations in the economical multifactor designs employed. Uniformity in the sizes of programs studied may also have limited these results. The range of values assumed by complexity metrics computed on these programs may have been insufficient for correlational tests to detect the strong relationships reported in other verifications of these theories. Studies reporting higher correlations for Halstead's E usually involved a broader range of program sizes. 8, 14

Further work in the area of software complexity should identify a set of cognitive principles relevant to programming tasks. Metrics could then be developed which would assess the qualities of software which are most closely related to these principles. Such an exercise might not only lead to improved metrics for assessing software complexity, but might also identify programming practices which could lead to more easily maintained software.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the assistance of Ann Fitzsimmons in implementing this research, of Dr. Gerald Hahn in developing the experimental design, and of Beverly Day in manuscript preparation. Careful reviews of this report by Dr. John O'Hare, Dr. Maurice Halstead, and Thomas McCabe have resulted in substantial improvements. The support and encouragement of Gerald Dwyer has also been greatly appreciated.

This study was supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N00014-77-C-0158). The views expressed in this paper, however, are not necessarily those of the Office of Naval Research or the Department of Defense.

Expanded reports of each experiment can be obtained by writing the senior author at: General Electric, Suite 200; 1755 Jefferson Davis Hwy.; Arlington, VA. 22202. Portions of this paper were drawn from articles to be published by
the Human Factors Society and the Institute of Electrical and Electronics Engineers.

REFERENCES

2. Cornell, L., and M. H. Halstead, Predicting the number of bugs expected in a program module (Tech. Rep. CSD-TR-205), West Lafayette, IN, Purdue University, Computer Science Department, October 1976.
17. Halstead, M. H., Using the methodology of natural science to understand software (Tech. Rep. CSD-TR-190), West Lafayette, IN, Purdue University, Computer Science Department, 1976.
22. Ottenstein, K. J., A program to count operators and operands for ANSI-FORTRAN modules (Tech. Rep. CSD-TR-196), West Lafayette, IN, Purdue University, Computer Science Department, June 1976.