INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University Microfilms International
300 N. ZEEB ROAD, ANN ARBOR, MI 48106
18 BEDFORD ROW, LONDON WC1R 4FJ, ENGLAND
BAKER, ALBERT LOUIS
SOFTWARE SCIENCE AND PROGRAM COMPLEXITY MEASURES.
THE OHIO STATE UNIVERSITY, PH.D., 1979

© 1979

ALBERT LOUIS BAKER

ALL RIGHTS RESERVED
SOFTWARE SCIENCE AND PROGRAM COMPLEXITY MEASURES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Albert L. Baker, B.A., M.S.

* * * * *

The Ohio State University
1979

Reading Committee:
Dr. Stuart H. Zweben
Dr. H. William Buttelmann
Dr. Anthony E. Petrarca

Approved By

Adviser
Department of Computer and
Information Science
To Betty and Jared
whose love has helped keep this in perspective
ACKNOWLEDGMENTS

I must first acknowledge the love and patience of my wife and son who have had to help balance our eagerness to complete this work with our shared family devotion. Only those who have shared this dilemma can fully appreciate the sincerity of my thanks to them. Then I offer my sincere gratitude to my research adviser, Dr. Stuart H. Zweben, whose constant guidance and insight have made this final product possible. His diligence for sharing in this work has been invaluable.

Next I feel confident that I can express a heartfelt thanks from Dr. Zweben and myself to Dr. H. William Buttelmann for his guidance through the latter stages of my Ph.D. program. We also acknowledge the advice of the other member of my reading committee, Dr. Anthony E. Petrarca.

I would also like to thank my other friends who have lent both personal and professional assistance. This large group includes Mike, Chuck, and Ellen. It also includes fourth floor comrades Wayne, Paul, Doyt, and Natarajan. I know that in each of these wonderful people I have a friend for life.

The valuable opportunity to edit much of what appears in this dissertation on the l.R.C.C.'s Dec-10 computer is also acknowledged, as is the partial financial support provided by Dow Chemical, U.S.A. under grant number 3527.

Finally, I wish to express my appreciation as a grateful son to my loving parents, Fred and Wilda.
VITA

September 29, 1949..... Born--San Francisco, California
August 1974............. B.A. Math., Drake University, Des Moines, Iowa
1974--1975............. Teaching Associate, Department of Mathematics, The Ohio State University
1975--1978............. Teaching Associate, Department of Computer and Information Science, The Ohio State University
December 1975........... M.S. C.I.S., The Ohio State University, Columbus, Ohio
1978--1979............. Research Associate, Department of Computer and Information Science, The Ohio State University

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science
Areas of Study:
Software Engineering. Dr. Stuart H. Zweben
Computational Linguistics. Dr. H. William Buttelmann
Information Storage and Retrieval. Dr. Anthony E. Petrarca
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. THE OBJECTIVE: QUANTIFY PROGRAM COMPLEXITY</td>
<td>1</td>
</tr>
<tr>
<td>1.1. The Programmer-Program Relationship</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Evaluating Program Complexity Measures</td>
<td>12</td>
</tr>
<tr>
<td>1.3. Dissertation Overview</td>
<td>17</td>
</tr>
<tr>
<td>2. RESEARCH PERTINENT TO THE OBJECTIVE</td>
<td>21</td>
</tr>
<tr>
<td>2.1. Empirical Studies</td>
<td>24</td>
</tr>
<tr>
<td>2.1.1. Production Environment Studies</td>
<td>24</td>
</tr>
<tr>
<td>2.1.2. Controlled Experiments</td>
<td>26</td>
</tr>
<tr>
<td>2.1.3. Experimental Control Through Language Design</td>
<td>35</td>
</tr>
<tr>
<td>2.2. Software Science</td>
<td>46</td>
</tr>
<tr>
<td>2.2.1. General Development</td>
<td>48</td>
</tr>
<tr>
<td>2.2.2. The Counting Strategy</td>
<td>62</td>
</tr>
</tbody>
</table>
2.2.3. Evaluations of E........................................68
2.3. Control Flow Complexity Measures...............85
  2.3.1. Program Knots.................................85
  2.3.2. Normal Number...............................87
  2.3.3. Cyclomatic Number............................88
3. SOFTWARE EFFORT AND MODULARITY..................97
  3.1. Common Subexpressions..........................102
  3.2. Simple Common Subsequences...................107
  3.3. Common Subsequences.............................136
  3.4. The Counting Strategy..........................142
  3.5. Summary of Modularity Analysis................148
4. EVALUATION OF MEASURES OF CONTROL
FLOW COMPLEXITY........................................149
  4.1. SOFTWARE EFFORT..................................150
    4.1.1. THE LINEARIZATION ISSUE....................150
    4.1.2. THE NODE SPLITTING ISSUE...................156
    4.1.3. STRUCTURING MULTIPLE EXIT LOOPS...........175
  4.2. PROGRAM KNOTS....................................179
  4.3. NORMAL NUMBER....................................185
  4.4. CYCLOMATIC NUMBER................................187
5. A SYNTHESIZED PROGRAM COMPLEXITY MEASURE.........205
  5.1. A First Attempt..................................206
  5.2. Limiting the Domain to Define PC..............214
  5.3. An Evaluation of PC..............................229
6. SUMMARY: THE OBJECTIVE IS CLOSER..................234
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Relationships in Program Processing</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>The Basis of Analytical Evaluations</td>
<td>16</td>
</tr>
<tr>
<td>3.</td>
<td>Program Segment Containing One Knot</td>
<td>86</td>
</tr>
<tr>
<td>4.</td>
<td>Flowgraph</td>
<td>89</td>
</tr>
<tr>
<td>5.</td>
<td>Normal Version of Flowgraph in Figure 4</td>
<td>89</td>
</tr>
<tr>
<td>6.</td>
<td>Flowgraph with Edges Numbered</td>
<td>92</td>
</tr>
<tr>
<td>7.</td>
<td>Basis and Corresponding Vectors for Flowgraph in Figure 6</td>
<td>92</td>
</tr>
<tr>
<td>8.</td>
<td>Alternative Constructs of Varying Complexity</td>
<td>95</td>
</tr>
<tr>
<td>9.</td>
<td>Program Containing Two Simple Common Subsequences</td>
<td>111</td>
</tr>
<tr>
<td>10.</td>
<td>Disposition of Operands in Subprogram Definition of a scs</td>
<td>119</td>
</tr>
<tr>
<td>11.</td>
<td>Revision of Program in Figure 9</td>
<td>120</td>
</tr>
<tr>
<td>12.</td>
<td>Flowgraph</td>
<td>152</td>
</tr>
<tr>
<td>13.</td>
<td>Linearization of Flowgraph in Figure 12 with Three GOTO's</td>
<td>152</td>
</tr>
<tr>
<td>14.</td>
<td>Linearization of Flowgraph in Figure 12 with Four GOTO's</td>
<td>153</td>
</tr>
<tr>
<td>15.</td>
<td>Flowgraph Construct to be Linearized as DO...WHILE</td>
<td>155</td>
</tr>
<tr>
<td>16.</td>
<td>Flowgraph</td>
<td>158</td>
</tr>
<tr>
<td>17.</td>
<td>Tree Corresponding to Flowgraph in Figure 16</td>
<td>158</td>
</tr>
</tbody>
</table>
Figure 36 which Contains One Knot ........................................ 180
38. Linearization of Flowgraph in Figure 12
which Contains Two Knots ............................................... 181
39. A Different Linearization of Flowgraph in
Figure 12 which also Contains Two Knots ......................... 182
40. FORTRAN Implementation of IF ... THEN ... ELSE
with One Knot .......................................................... 183
41. Algorithm for Generating a Depth First
Spanning Tree ............................................................. 188
42. Flowgraph ............................................................. 189
43. Spanning Tree and Added Arcs for Flowgraph
in Figure 42 ............................................................... 189
44. Alternate Node Splitting Procedure
STRUCTURE ............................................................. 194
45. D as Node Splitting Candidate ..................................... 197
46. The Result after D is Split in Flowgraph in
Figure 45 ................................................................. 197
47. Multiple Exit Loop ................................................... 199
48. Structured Multiple Exit Loop ..................................... 200
49. Flowgraph Representation of an Alternative
Construct ................................................................. 203
50. Flowgraph Representation of an Iterative
Construct ................................................................. 204
51. Flowgraph ............................................................. 209
52. Spanning Tree and Added Arcs for Flowgraph
in Figure 51 ............................................................... 209
53. One Spanning Tree for a Flowgraph ............................. 212
54. Another Spanning Tree for the Flowgraph in
Figure 53 ................................................................. 212
55. Spanning Tree for the Iterative Construct .................. 217
56. One Spanning Tree for the Alternative
Construct ................................................................. 218
57. The Second Spanning Tree for the Alternative Construct .................. 218
58. Spanning Tree for an Iterative Structure Nested in an Iterative Structure ........... 219
59. Spanning Tree for an Alternative Structure Nested in an Iterative Structure .......... 219
60. Spanning Tree for an Iterative Structure Nested in an Alternative Structure ........... 220
61. Spanning Tree for an Alternative Structure Nested in an Alternative Structure ........... 220
62. Flowgraph Decomposed into Single Entry, Single Exit Subgraphs .................. 223
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Effect of Common Subexpression Reduction on Software Volume</td>
<td>107</td>
</tr>
<tr>
<td>2.</td>
<td>Effect of Reducing Simple Common Subsequences with Two Parameters and No</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Local Variables</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Effect of Reducing Simple Common Subsequences with Four Parameters and No</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>Local Variables</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Effect of Revised Counting Strategy for Common Subsequences Requiring Two</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Effect of Revised Counting Strategy for Common Subsequences Requiring Four</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Comparison of Performance of PC and E</td>
<td>233</td>
</tr>
</tbody>
</table>
CHAPTER I.

THE OBJECTIVE : QUANTIFY PROGRAM COMPLEXITY

Computer software production is evolving from an individualistic artistic endeavor toward a sound engineering discipline. That programming began, and existed for many years, as an essentially undisciplined art form is strikingly supported in the title of one of the most widely referenced computer science works—*The Art of Computer Programming* by D. E. Knuth [KHU68]. Additional support for this premise is supplied by the frequently repeated characterization of programming in the early years as "a cottage art" or "cottage industry". However, natural selection is beginning to take its toll on the species of solitary coders with the advent of mounting pressures for a better software product. These pressures have been characterized as "the software crisis", a widely used phrase which denotes a problem described by B. W. Boehm in [BOE73]. Boehm substantiates the hypothesis that software costs have
LIST OF TABLES

Table                      Page
1. Effect of Common Subexpression Reduction on Software Volume ...........................................107
2. Effect of Reducing Simple Common Subsequences with Two Parameters and No Local Variables ..............................................133
3. Effect of Reducing Simple Common Subsequences with Four Parameters and No Local Variables ..............................................134
4. Effect of Revised Counting Strategy for Common Subsequences Requiring Two Parameters ..........145
5. Effect of Revised Counting Strategy for Common Subsequences Requiring Four Parameters ..............................................146
6. Comparison of Performance of PC and E' ..............................................233
been rising much faster than hardware costs. This trend has paralleled, and been fostered by, the discrepancy between the existing state of the art of programming and the actual requirements placed on software developers. R. W. Jensen and C. C. Tonies advocate the use of the word "exigence" to describe the software crisis, because of this difference between current software technology and the technology demanded by today's applications [JEN79]. Another way of viewing this exigence is that computer hardware advances have significantly increased the complexity of the problems for which computer scientists might reasonably be expected to provide solutions. But while the hardware has advanced rapidly, the software tools have evolved more slowly. This is the result of the intellectual unmanageability of large software systems. If one were to consider a computer operating system to be a machine, then this "machine" would be one of the most complex devices man has constructed. Yet, unlike other builders, the software designer has had woefully few tools to aid him in his undertaking.

In order to reduce this commonly acknowledged exigence, a discipline of software engineering has been developing over the last ten years. Within this discipline efforts have been made to enumerate the desirable properties of quality software and to provide programming methodologies and language features which can be readily applied to the
production of this quality software. However, in order for this undertaking to result in a true engineering discipline, it is essential that measures be developed for software analysis. Perhaps the first request for such measures came from J. Sammet in 1969 [SAM69]. The implication of the request is that it is no longer sufficient for a well-known computer scientist to rely on her or his reputation and subjective judgment to substantiate a particular language feature or programming methodology. Software engineers must strive to support quantitatively their claims about particular software devices.

1.1. The Programmer-Program Relationship

The following simple model of the relationships involved in program processing helps to identify where deficiencies exist in our ability to quantitatively evaluate computer software. Most of the research concerned with software analysis has dealt with the relationship between the program and the machine which executes the program. Measures exist for dealing with storage and timing considerations at many levels of abstraction of the notion of "machine". On the other hand, little work has been directed at increasing understanding of the complex relationship between the programmer and his or her
software. The software crisis is the immediate result of this lack of understanding.

What programming activities actually fall within this notion of the programmer-program relationship? Are there reasonable measures of these activities? One obvious programming activity is program generation and several readily obtainable measures quantify this activity:

1. Programming time. Researchers who have used programming time to measure program generation have varied the components of the measure, but in its broadest interpretation it includes all design time given an informal problem specification, coding time, time spent verifying the code (i.e. desk-checking), and testing and debugging time (e.g. see [ZIS73]).

2. Number of bugs during program development. An
example of research which actually uses this measure of the programming activity is F. Akiyama's analysis of a system's development by inexperienced programmers [AKI72].

3. In a step-wise refinement design process, the number of steps generated for each refined instruction. This measure of program generation is utilized in a system developed by R. W. Witty [WIT78] for Rutherford Laboratory in Great Britain. A design process called dimensional flowcharting is totally reflected in the resulting code in this Fortran-based system. Since the design process appears in the final code, this measure is accurately obtainable in Witty's system.

4. Number of rejected possibilities in the design process.

Another aspect of the programmer-program relationship which is at least as common as program generation includes those tasks which involve working on existing code. Such tasks may be adequately characterized as program maintenance. However, obviously adequate and readily obtainable measures of this activity are not as easy to identify as they were for program generation. Some measures which have been used to try to quantify these activities are:
1. Data from rote memory experiments. T. Love used the percentage of correctly recalled statements after a fixed study time in experiments on various program features [LOV77]. Similar data and experiments were considered by Shneiderman [SHN76].

2. Hand execution time. This measure of the programmer-program relationship was used by L. M. Weissman in experiments dealing with the notion of complexity of computer programs [WEI74].

3. Measures of responses given to questions concerning the code. The approach used to obtain this measure was suggested by G. M. Weinberg [WEI71]. It involves permitting subjects a fixed timespan to study a given code segment and then posing a set of questions concerning the studied code. Shneiderman also performed experiments using this type of measure [SHN76].

4. Time to find a given bug. This measure was obtained in experiments performed by J. D. Gould in which subjects were asked to discover a planted bug in given code segments [GOU72].

It must be reiterated here that some of these measures of the activity of program maintenance may not as fully capture their intended subject as do the previously enumerated measures of program generation. This is the result of the
fact that none of the suggested measures of program maintenance directly measures the activity—they measure a secondary activity (e.g. answering questions about a program) which is (hopefully) related to program maintenance.

These examples of measures of program generation and maintenance all provide some quantification of the difficulty the programmer encounters when he or she works with software. Not that any one of the measures alone captures this abstract notion of difficulty, but they at least, taken as a set, help to characterize the concept. In order to avoid confusing this idea with related ideas which are developed later, the following definition is provided.

**Definition:** The abstract notion of the difficulty incurred for the programmer when he or she produces or maintains a given software segment is called **programming difficulty.**

Although the measures of program maintenance activities are more difficult to obtain because they are derived from secondary activities, both generation and maintenance measures suffer in their utility to the software engineer in that they must in some sense be obtained in controlled experimental situations—they are not usually available from
a finished software project as a natural result of the programming activities. Thus these measures will not receive widespread use for evaluating large, arbitrarily chosen sets of programs. The measures do not satisfy J. Sammet's request for a measure of the actual code which captures a major portion of the notion of programming difficulty. Thus, what we seek is the following:

**Definition**: A program complexity measure is a measure, obtainable from the actual program source code, which captures some commonly accepted aspect of programming difficulty.

The requirement that the measure be obtainable from source code will give such measures the utility lacking in the previously enumerated measures of programming difficulty.

A broader intuitive grasp of program complexity is obtained by considering the three sources of program complexity. Some program complexity is inherent in the problem solved by the program. To understand this inherent complexity more clearly, imagine there exists an oracle which when given a particular problem will spew out a correct algorithmic solution to that problem. Also assume that the oracle's solution minimizes programming difficulty by the combination of the language chosen to express the solution and the algorithm itself. No other solution in any
language can do more to reduce programming difficulty. Then two different problems whose ideal solutions have different programming difficulty have inherent complexity differences. A second source of program complexity lies in the selection of less than ideal algorithms. It is a safe bet that the oracle incarnate has never existed. The final source of program complexity comes from the implementation of an abstract algorithm in an actual program. For example, there may be several ways of expressing a two-dimensional flowgraph as a sequential set of statements in a programming language. Again relying on a "higher authority" with perfect knowledge of programming difficulty, it must be the case that one of these "linearizations" will minimize programming difficulty. Clearly, each of the sources of program complexity will have a direct effect on programming difficulty.

On the other hand, there are aspects of programming difficulty which cannot be reflected in the final source code, and so cannot be reflected in a program complexity measure. Such sources were alluded to by the fourth measure of program generation, the number of rejected possibilities in the design process. While such backtracking necessarily incurs programming difficulty, it cannot be reflected in any measure of the resulting source code. This poses something of a dilemma for empirical research on program complexity
measures. One can correctly infer from the foregoing discussion that a measure of program complexity will most accurately reflect programming difficulty associated with the activity of program maintenance. However, as was noted previously, solid measures of this programming activity are in some ways deficient. So, in any experimental evaluation of a proposed measure of program complexity, one is faced with two less than perfect alternatives. The first is to compare the measure against measures of program generation, knowing that generation measures may reflect programming difficulty which cannot be captured by the proposed measure. The second alternative is to compare the proposed measure with measures of program maintenance, knowing that these measures are in some ways inadequate. R. Gordon in his research efforts on a program complexity measure has wrestled with exactly this dilemma. Some discussion of his resolution can be found in [GOR79A].

Thus the primary objective of this research is motivated: to contribute significantly to the development of a good measure of program complexity. The primary impetus for such a measure, as has been discussed, is its utility in putting software engineering discussions of language features and programming methodologies on considerably firmer scientific ground. But one can envision additional applications for an established measure
of program complexity. One such application is the result of the fact that programmer managers are simply not able (even those who have the technical expertise) to review and evaluate a significant fraction of the code produced by their employees. A good measure of program complexity could aid a manager in spotting those programmers who seem to regularly produce programs with greater complexity. If the manager has sound reason to believe that these programmers are not regularly receiving problems with the greatest inherent complexity, then he may wish to review in detail code produced by these workers.

This application of a good measure of program complexity by managers suffers from the weakness that it must ultimately be based on an evaluation of the inherent problem complexity. As yet there do not exist good objective measures of problem complexity, so that the use of the measure would be based on a subjective judgment. However, the use of a good measure of program complexity in the classroom would not suffer from this drawback—many students solve the same problem. What computer science educator does not loathe the task of grading student programs? A good program complexity measure could be used as a tool to flag those student programs which seem to have problems—not from a correctness standpoint, but rather from the point of view of programming difficulty. Those students
(hopefully, a majority) whose programs fall into a roughly equal, relatively low range of program complexity could safely be afforded far less attention.

1.2. Evaluating Program Complexity Measures

The motivation of the research objective leaves two questions which should be addressed in a complete introduction to research in program complexity measures. These are, "What are the measurable properties of software which might serve as program complexity measures?" and, "What methods are available for evaluating proposed program complexity measures?". Detailed discussion of the answer to the former question is delayed for Chapter 2, but a few examples at this point may be illustrative. The measurable properties of software fall into either a general category or one consisting of measures related to program control flow. A few examples of the general measures are:

1. Number of lines of source code.
2. Number of program statements.
3. Operator and operand counts. All software science measures are based on such counts and will be described in considerable detail in Chapter 2.
4. Measures of the "distance" in source program text over which individual variables are referenced.
5. Measures of program form. Such measures usually consist of a determination of whether or not a coding convention has been followed. These conventions include:
   a. Source code formatting (e.g., logical indentation).
   b. Meaningful identifiers.
   c. Adequate program documentation.

The following are examples of control flow complexity measures:

1. Number of possible execution paths in the program. This measure is only applicable to structured programs. A definition of such programs and a discussion of this observation are presented in later chapters.

2. Cyclomatic number. This graph theoretic measure, which was initially suggested as a program control flow complexity measure by T. J. McCabe [MCC76], is investigated in detail in Chapter 4.

3. Number and type of transfer of control operators.

4. Number of referenced labels.

5. Quantification of nesting depth of statements.

Clearly, the two preceding lists provide but an initial insight into the type of measure which might serve as the desired program complexity measure. Much of what follows
will be a thorough examination of the more promising possibilities.

But, before any measures of program complexity can be evaluated, the various approaches to such evaluation should be considered. The possible evaluation methodologies may be classified into two groups. The first involves testing the proposed measure empirically, an approach which has already been alluded to. Investigations of this type consist of comparing the proposed program complexity measure with one or more of the measures of either generation or maintenance activities. In other words, an attempt is made to determine the extent to which the proposed measure satisfies the definition of a program complexity measure by using the measures of programming activity as indicators of programming difficulty. An example of this type of evaluation is R. D. Gordon and M. H. Halstead's studies of the software science effort measure, E, (see Chapter 2 for a detailed discussion of this measure and the study) [GOR76]. These researchers compared the behavior of E with actual programming time.

A second approach to evaluating program complexity measures is more analytical in nature. The basis of comparison in this case is a well-defined source-to-source program transformation which embodies some principle of
programming methodology about which there is consensus concerning its utility. To see this more clearly suppose that, in Figure 2, $T$ is a source-to-source transformation and there is considerable agreement that $T(P)$ is a more intellectually manageable algorithm which performs the same function as the original program $P$. As long as $T$ is well-defined, it should then be possible to determine analytically the behavior of the proposed program complexity measure under $T$, resulting in an expression for the measure on $T(P)$. Then the measure on $T(P)$ can be compared with the measure on $P$—the measure on $T(P)$ should be smaller. That is, the measure ought to agree with the consensus. It might be worth pointing out that such analytical evaluations make use of precisely those rather subjective evaluations of programming principles which a good program complexity measure should eventually eliminate. But there are various principles which the discipline as a whole has adopted (and appropriately so) and, in order for any measure of program complexity to receive wide acceptance, it will therefore have to agree, for the most part, with these accepted principles. The measure will have to stand up to this type of analytical investigation. It is also worth pointing out that this type of analysis does not imply that the transformations are best applied in an ex post facto fashion. Both the general consensus and a good program
complexity measure ought to support principles which should be applied by the programmer as he or she develops software.

\[ T_P \rightarrow T(P) \]

The Basis of Analytical Evaluations

Figure 2

Despite its importance, it is this type of evaluation which has been for the most part ignored. One study which touches on an aspect of this analytical methodology is the previously cited work by Gordon on the software science effort measure, \( E \), as a measure of program complexity [GOR79A]. The evaluation involved specific examples of program improvements which had been supported subjectively in the open literature by programming authorities. However, Gordon's work is essentially empirical in that his intended purpose was not to establish the behavior of \( E \) under general programming principles, but rather to illustrate its good behavior with specific examples. In a separate evaluation of \( E \), Gordon does use an analytical approach by showing that \( E \) is reduced when certain undesirable program constructs are removed [GOR79B]. These constructs have been labelled
"program impurities" [HAL73A]. They are described fully in Section 2.2. However, given that but one analytical evaluation of E has been performed, it is certainly the case that it is precisely this type of analysis which is required to bolster confidence in E.

1.3. Dissertation Overview

The succeeding chapters contain a complete description of a research effort which contributes to the development of a program complexity measure through the use of analytical evaluations. In Chapter 2 the software science effort measure, E, is fully developed and research aimed at establishing E as a measure of program complexity is reviewed. In addition, Chapter 2 contains descriptions of other research pertinent to reaching the objective of a program complexity measure. In Chapter 3, E is evaluated analytically using certain principles of program modularity. Specifically, the behavior of E is generalized in the cases of:

1. reduction of common subexpressions,
2. implementation of repeated, similar code sequences as subprograms whose arguments are simple variables or constants, and
3. subprogram definition without the argument
restriction.
In all these cases data flow analysis is used to provide sound definitions of the environments in which the transformations may be performed. The overall summary of this evaluation of E and modularity concepts is that the behavior of E agrees strongly with widely held principles concerning program modularity. Thus, within the scope of the modularity analysis, E appears to be a strong candidate for the desired program complexity measure.

The next analytical evaluation of E uses principles related to program control structures and is described in Chapter 4. In this case, while there is one major area in which E performs well, there are other areas where the behavior of E poses some problems. This leads to the conclusion that, while the research of Chapter 3 and previous work described in Chapter 2 strongly support E as a good measure of program complexity, E is unable to adequately capture the complexity incurred by program control flow.

For this reason, other approaches to the quantification of program control-flow complexity are also studied in Chapter 4. One promising measure, T. J. McCabe's cyclomatic complexity measure, C, [MCC76], is considered in detail. An important theorem which goes a long way toward
justifying this graph theoretic measure as a measure of program control flow complexity is stated and proved. Several properties concerning the effect on $C$ of composing structured programs from basic structured constructs are established, and then two theorems characterizing the effect on $C$ of two structuring transformations are stated and proved. These analytically derived results contribute significantly to confidence in $C$ as a measure of control flow complexity.

The results of Chapters 3 and 4 give rise to the following situation. $E$ satisfies all the tests to which it has been subjected as a proposed measure of program complexity, except certain tests related to program control flow complexity. The analysis of $C$ indicates that it is the best measure of control flow complexity (though it too has a few identifiable weaknesses) and thus quantifies those areas in which $E$ is weak. Chapter 5 addresses the obvious question, "Can the desirable properties of $E$ and $C$ be merged in a new measure of program complexity which satisfies the analytical problems which arise when either $E$ or $C$ is taken alone?" An answer to this question begins with proofs of several additional theorems concerning cyclomatic complexity. These theorems indicate that a well-defined measure synthesizing $E$ and $C$ is readily obtainable for structured programs. This new measure of program complexity
is denoted PC and the derivation of PC establishes analytically that the measure does in fact handle the previously enumerated problems with E or C. Finally, PC is subjected to empirical validation by using the data from the Gordon complexity experiment cited earlier.

The summary in Chapter 6 presents three areas where this research contributes to the overall objective of a measure of program complexity. The first is in the enhancement of E as a program complexity measure. The second is the contribution to confidence in the cyclomatic number as a measure of control flow complexity. Finally, the question of whether there is hope that E and C might be synthesized into a new measure of program complexity is answered in the affirmative. This final chapter concludes with some observations about additional research which is needed in the pursuit of a widely accepted measure of program complexity.
Quite recently, the rate of publication of research concerned with the programmer-program relationship has accelerated. This growing interest indicates that a significant number of computer scientists believe that increasing understanding of the programmer-program relationship can, in fact, help cool the software crisis. In this chapter three lines of research which have contributed to the base of knowledge of this relationship are reviewed in detail. Taken together, these three areas provide a background for the results presented in the subsequent chapters.

The first set of research efforts can be classified as empirical studies of programs and programming activities. Some early works of this variety indicated the types and frequencies of errors made in programming. But G. M. Weinberg’s *A Psychology of Computer Programming*
[WE71] was a pioneering indicator of the need to identify those software features and approaches to software production which might reduce programming difficulty (of which the error frequencies in the early experiments are measures). This need that Weinberg made explicit motivated numerous experiments in which measures of program generation and program maintenance were used as measures of programming difficulty to evaluate language features and programming methodologies. The review of these empirical studies contained in Section 2.1 is pertinent to program complexity measure research for several reasons. The studies provide initial insight into the programmer-program relationship. They also contain numerous approaches to quantifying programming difficulty and thus have been used in evaluations of program complexity measures. In addition, the limitations of such experiments (which will become apparent in the review) provide considerable motivation for the development of program complexity measures.

Section 2.2 contains a summary of research spawned by M. H. Halstead's unique philosophical approach to investigations of the programmer-program relationship. He believed that both the preparation and ultimate form of written materials are governed by natural laws which may be substantiated by classical empirical science [HAL77]. The research generated by this philosophy is now denoted as
software science. One result contained in this work is the measure $E$, derivable from the source code, which purportedly quantifies the "mental effort" spent in generating or maintaining a given program. Thus Halstead's "mental effort" is exactly programming difficulty, and $E$ is a program complexity measure. Specifically, Section 2.2 surveys the major contributions of software science research to knowledge of the structural properties of computer programs, and presents in detail the existing studies of the program complexity measure $E$.

In the final section of the literature review, other program complexity measures are presented. Chapter 1 contains an indication that program complexity measures which are specifically tuned to quantifying program control flow complexity may be helpful in synthesizing a new program complexity measure. The measures introduced in Section 2.3 are of this variety, and will be the subject of analysis in Chapter 4.
Section 2.1. Empirical Studies

2.1.1. Production Environment Studies

The earliest efforts to empirically study programs and program generation are characterized by their goal of identifying and classifying program errors and bottlenecks in program generation. Thus they deal with actual production programs and widely used, general purpose programming languages. Rubey [RUB68] performed such studies on programs written in one language, while other early studies concentrated on analysis of the batch versus time-sharing controversy (e.g. [ADA69] and [GRA67]). Two of the early studies which are of particular interest because of their results on the nature of program errors are described in a little more detail, as they are fairly representative.

E. A. Youngs in "Human Errors in Programming" [YOU74] made it clear that his work is concerned with the programmer-program relationship when he emphasized his goal of studying programming as "human behavior". Youngs used answers to questionnaires, time logs and all output to obtain measures of programming difficulty. Errors were analyzed by working back from the output of the final run to that of the initial run. The analysis of the data from 69
programs showed that eight functionally defined constructions account for 75% of the errors committed. The most troublesome constructions were those involving assignment (29%) or allocation (16%). The analysis also indicates that the most persistent errors (those errors that remained uncorrected over the largest number of runs) are explicit transfer of control errors. Youngs' work is informative, but does leave a few pertinent questions unanswered. What would the distribution of a specific type of error (e.g. syntactic errors) look like? Does the choice of language affect the results? Is the sample size statistically significant? These results also indicate that general error rates may provide only a coarse measure of programming difficulty.

S. J. Boies and J. D. Gould also addressed the question of error types [BOI74]. This work again made use of programs written in a production environment. The data in this case was collected by an on-line system called SIPE which counts errors detected by various language processors. Analysis of data collected over two separate five day periods indicates that only about one out of five production runs contains errors detectable by the language processor. Boies and Gould believe that this indicates that extensive additional research on automated error correcting language processors is unwarranted, since the errors which
could be corrected by such systems do not constitute a programming bottleneck. In addition, this result may indicate that number of syntactic errors may not be particularly useful as a measure of programming difficulty.

These two studies when viewed together help to clarify one of the limitations of empirical studies. Although the Boies and Gould study did address one of the questions raised by Youngs' work, it did not clarify whether or not the error frequency distributions of Youngs' data are dependent on the type of error. It is possible that many of the errors in assignments are syntactic—the two studies are not mutually supportive. Of course, Boies and Gould may not have had access to Youngs' work since the studies were performed at roughly the same time, but this lack of mutual support will recur in the other studies presented in Section 2.1, and may well be the result of the difficulty of experimentally answering such questions—the number of experimental variables involved in production programs written in standard languages is extremely large.

2.1.2. Controlled Experiments

Clearly one way of limiting the proliferation of variables that troubles the analysis of studies of production programs is to much more rigidly control the experimental situation. Although the early studies just
cited point to a need for this controlled approach, it should be noted that there is a trade-off involved in using controlled experiments in that one can always question the extensibility of results obtained in these experiments to actual working environments. However the more recent works indicate that researchers believe the trade-off favors the controlled approach. Therefore a representative sample of the early controlled experiments is reviewed.

Following his conclusion about syntactic errors in programs, Gould continued his efforts to delineate the bottlenecks in programming by focusing attention on nonsyntactic errors in his work with P. Drongowski [GOU74]. This study limited the variables in question to the amount of information provided to aid debugging and the type of program bug. The experimental situation involved discovering exactly one bug planted in one line of a FORTRAN program. The measures of programming difficulty employed were the length of time required to identify the bug and the number of wrong guesses made in attempting to identify the bug. Thus the aspect of programming difficulty addressed resulted from program maintenance activities. The results of the study indicate that those aids one might intuitively consider helpful in debugging are of little value, and that errors in assignment are the hardest to correct. Finally, Gould and Drongowski were able to conclude that a "sweat and
blood" approach to program debugging may be as fruitful as any interactive debugging process. But the authors acknowledge that these conclusions are predicated by the short programs and the limited number and restricted location of program errors. The extensibility of the conclusions to real programming environments is questioned.

Another attempt to control the programming environment is reported by B. Shneiderman in [SHN76]. This work is one of the first programming research efforts to draw on the discipline of psychology to support at least some aspect of the experimental design. Shneiderman is also credited with explicitly connecting his empirical work with the type of research results presented in the later chapters of this dissertation. He made this connection when he gave as a long range goal of empirical studies the "creation of problem and program complexity measures." In the article, Shneiderman enumerates numerous measures of programming difficulty, but his choice of percentage of correctly recalled lines for his first experiment is supported by the notion of "chunking". This term, borrowed from psychology, denotes the mental process of organizing information into intellectually manageable units. The greater the extent to which this process can be applied, the greater the capacity to memorize a given sample. This hypothesis was tested by Shneiderman using subjects with varying levels of
programming experience by asking them to memorize either a shuffled or unshuffled version of a FORTRAN program. The percentage of correctly recalled lines supports the chunking theory and indicates that it is more applicable for more experienced programmers.

In a second experiment reported in the same paper, Shneiderman investigated the programming difficulty incurred by the FORTRAN logical IF as opposed to the arithmetic IF. The measures of programming difficulty utilized were: number of errors in answers to questions concerning program sequencing, number of errors in specifying the output produced by the program (given specific input values), and number of errors in fill-in-the-blank statements and multiple choice questions concerning the program's function. Both experienced and novice programmers were used as subjects. Shneiderman concluded from this experiment that experienced programmers have little difficulty handling either construct, but that novice programmers work more comfortably with the logical IF. The significance of this work as a contribution to the direction of this type of empirical research has already been covered, but it is worth pointing out that it also leaves unsettled the question of identifying adequate measures of programming difficulty. The measure used in the first experiment only reveals that programmers memorize unshuffled programs more readily than
shuffled programs. This does not substantiate memorization facility as a particularly refined measure of programming difficulty. The measures in the second experiment are all ex post facto measures of program understanding, and they do not have the support of an underlying theory of human cognition like chunking.

T. Love reports on another controlled experiment in [LOV77]. This represents one of the early attempts to apply the empirical approach to points of contention related to the structured programming controversy. Love agrees with Shneiderman's positive comments on the worth of memorization as a measure of programming difficulty and so used the percentage of correctly recalled lines as his actual measure. The experiment involved memorization exercises by experienced and novice programmers. The independent variables tested were program paragraphing and control flow. Simplified control flow was achieved by the use of D-structures (1) which are characterized by being single entry, single exit program constructs. Denoting these control structures as "simplified control flow" might indicate prejudice, since the purpose of the experiment was to test whether or not these structures do in fact simplify control. Love also required that the experienced volunteers submit written descriptions of the program's function. These descriptions were evaluated and ranked by the
experimenter. The findings of the experiment support the argument that paragraphing is not a significant contributor to decreased programming difficulty. The results on control structures are not decisive. When memorization performance is used to evaluate the two approaches, the D-structures may in fact be considered less complex. The data from the written descriptions do not support this assessment. Probably the most significant overall result of this work is that, even with what is, in comparison to some other empirical studies, a fairly tightly controlled experiment, Love feels that more control is needed to reduce the ambiguities of the results. In addition, the utility of memorization as an adequate measure is still not well established.

---

(1) "D-structures" was coined by H. F. Ledgard and N. Marcotty (LED75) and it stands for Dijkstra-structures. A D-structure is constructed from the following:

a. Basic actions (e.g. assignment, I/O).
b. Composition of 2 D-structures s1 and s2 as s1; s2.
c. Alternative constructs: IF P THEN s1 ELSE s2, where P is a predicate and s1 and s2 are D-structures.
d. Iterative constructs: WHILE P DO s1 where P is a predicate and s1 is a D-structure.

This is a common formal definition of "structured program", and when the term "structured program" is used in this work it will be intended to denote programs constructed in this manner.
The final study which fits into this category of experiments which are more controlled than the production environment studies is that of L. M. Weissman [WEI74]. Although this work was reported before the other three studies which have just been reviewed, it is presented last because its original objective has not been achieved to date— the development of a sound experimental methodology for the conduct of this type of empirical study. The number of relevant questions which remain unresolved by these experiments substantiates the need to continue to pursue an adequate methodology.

In [WEI74], Weissman traces a sequence of five experiments and shows the weaknesses and strengths of each experiment. In the first experiment Weissman tried to test the effect of the presence or absence of documentation, paragraphing, and informative identifiers. In the experiment undergraduate subjects were evaluated for program understanding using how well the subject hand simulated a program, how well she or he was able to fill in the blanks in a paragraph describing the program, and a self evaluation. The principle implication of the first experiment is that students with less than one year of programming experience are unsuited for such studies (a finding which is, unfortunately, ignored frequently in subsequent studies). This result does support
G. M. Weinberg's caution against allowing the psychology of programming to be solely the psychology of student programmers (WEI71).

Weissman repeated the first experiment using more experienced subjects. In this second experiment statistically significant results were obtained. However Weissman found wanting the measures of programming difficulty used in the first two experiments and so these measures are modified in the subsequent experiments. In the third experiment more interesting program factors related to programming difficulty were studied and Weissman encountered a difficulty which has been alluded to—it is difficult to vary single factors of interest. The factors tested in the third experiment were control structures and paragraphing. As indicated, the measures of programming difficulty were also modified for experiment three, reflecting the deficiencies with those used in experiment two. The subjects were evaluated for program understanding (which resulted from reading and hand simulating a program) by a series of quizzes and self evaluations. The major implication of this experiment for a general experimental methodology is that the number of subjects may have to be quite large in order to obtain statistically significant results, and allowing subjects to proceed at their own pace may result in incomplete data.
Thus the fourth experiment was designed to overcome the difficulty of not structuring the subject's time. In this experiment the factor affecting programming difficulty which was tested was one notion of modularity. Again, Weissman found that his number of subjects was too small to yield statistically significant results, but he felt that his measures of programming difficulty may be adequate. The subjects were given 15 minutes to study the program and then 10 minutes for the first self-evaluation and quiz. Then 30 minutes were allotted for the hand-simulation of the program and 10 minutes for a second self-evaluation and quiz. In addition to the procedure of experiment three, the subjects were then asked to make a specific modification to the program in 30 minutes and then take 10 minutes for a third self-evaluation and quiz. It seems that the only drawback to this method of measuring how well the subject understands the program is the difficulty of evaluating the modifications requested in the final step.

In a fifth and final experiment Weissman questioned the need to hand simulate the programs. Three groups were given the following different instructions for the 30 minute period for studying the program: hand simulate the program, study a machine generated source language trace of the program, and study the program by any technique deemed appropriate. This experiment revealed no difference in the
performance of the subjects using any of the methods. Thus hand simulation may not be an important part of the methodology.

In summary, Weissman attempted to improve the experimental methodology while he tested a few programming principles. While Weissman's work lends some insights into how to conduct such experiments, it would seem from the other experiments reviewed that this need has not diminished significantly. How many subjects would be required to give statistically significant results in Weissman's experiments? Even if Weissman's measures of understanding (program maintenance related) are adequate, what are good measures of program generation? Can research from another discipline (e.g. psychology) be used to support the use of these programming difficulty measures? How can appropriate factors which are hypothesized as factors of programming difficulty be sufficiently isolated to permit adequately controlled experiments?

2.1.3. Experimental Control Through Language Design

In this final part of Section 2.1 two sets of controlled experiments are reviewed separately from the previous four works because they represent a different approach to the study of language features—an approach which has considerable appeal in terms of experimental
control. Both works use the programming language to control the language features to be investigated. This has the obvious advantage of rather tightly controlling the experimental variables in question while (potentially) allowing the subjects to work in production-like environments. The principal drawback of this methodology is the large overhead involved in writing a language processor and the associated documentation for each language to be tested. Even if this is actually accomplished by modification to existing systems, the work involved is significant.

The first studies using source language to control the experimental independent variables were those conducted by M. E. Sime and a number of colleagues. A series of experiments were performed using the same experimental tool. A simple programmable system called "the hungry hare" was developed and is described in detail in [SIM73]. These researchers believe that it is important to use untrained subjects in studies of language features so that prior training does not influence how the subjects perform using certain language features. (This premise may have some validity, but results using subjects who are competent programmers using the features in question may not agree with results obtained for those same programmers when they were novices.) Thus the Hungry Hare system was designed to
be used by nonprogrammers.

The initial experiment, which is reported in the paper cited above, made use of two languages for expressing instructions for Hungry Hare. The first language, called JUMP, contains an alternative construct which is FORTRAN-like in that what follows the predicate is an explicit transfer of control—a single statement. The other language, NEST, contains an ALGOL-like alternative construct. Neither language allows program loops. Two groups of "programmers" were used, one group for each language. Several subjects in the JUMP group failed to solve the specified problems while all of the subjects in the NEST group succeeded. Several other measures of programming difficulty were also obtained and each indicated a superiority of NEST over JUMP. In fact, when the two groups exchanged languages the various measures of programming difficulty indicate that the NEST group had a hard time learning JUMP while the JUMP group performed relatively well when allowed to program in NEST.

Sime, et. al., acknowledged two possible drawbacks to their experiment. First, the instructions issued to each subject on how to use each language may have biased the results. Second, the languages used may have been overly restrictive in that they support only one action per program
execution. A second study, [SIM77], addressed these drawbacks and extended the study of alternative constructs. An experiment was performed in which the problems used in the first Sime study were used again, only this time subjects chose from three possible solutions to a problem. One half the subjects chose from alternatives written in JUMP and the others from alternatives written in NEST. Only one correct program was present in each set of three. The results of this experiment were analogous to those in the case where the subjects did the programming, so the instructions on how to program were most likely not prejudicial.

Another experiment was designed to address the second drawback of the first Sime study—language limitations. Three languages were developed which allow multiple actions to be specified by a single program. The languages were also intended to facilitate an even more refined experiment on alternative constructs. One language was JUMP from the earlier study. A second called NEST-BE contains an ALGOL-like construct of the form:

IF P THEN BEGIN s1 END ELSE BEGIN s2 END

The third language, NEST-INE, handles the scope marking of the THEN and ELSE clauses in a slightly different manner:
IF P THEN s1 NOT P s2 END P

In both cases s1 and s2 may be composed of several statements.

In the experiment subjects with no prior programming experience learned one language. The measures of programming difficulty which were determined for each subject are number of semantic errors per problem, number of syntax errors, number of correct programs on first run, average number of runs to correct an error, length of time to solve the set of problems, and average length of time to complete a problem.

The authors conclude from the results for these measures that while the two "nest" languages are semantically superior for program generation, the NEST-BE language is prone to trivial errors. Of course Boies and Gould's conclusion that syntax errors are relatively unimportant would minimize differences between NEST-BE and NEST-INE on a purely syntactic basis, but the significantly shorter error lifetimes of errors in NEST-INE programs is important. This measure indicates that debugging is easier in NEST-INE programs, and hence gives this language some semantic, as well as syntactic, superiority. Sime, et. al., speculate that this debugging ease of NEST-INE programs
indicates that it is easier to "read backwards" in these programs, i.e. it is easier to determine the predicate values which produce a particular action. Program information which facilitates this working backward is called "taxon information", as opposed to "sequence information" which facilitates tracing a program's execution. In an experiment reported in [GRE77], T. R. Green established the superiority of the taxon information in NEST-INE over that in NEST-BE.

The one lingering programming methodology question from this experiment on three micro-languages is "Can the trivial errors of the NEST-BE language be reduced?" The importance of this question stems from the fact that the amount of redundant information in the NEST-INE IF statement would be unwieldy, especially in cases where the predicates are long. An experiment on possible ways to reduce these trivial errors was conducted by Sime and his colleagues and reported in [SIM77]. The experiment again used the hungry hare system. But in this experiment three different procedures for writing NEST-BE programs were provided. One was an automatic system which utilized a graphics display to automatically generate the statement form:

```
IF P THEN BEGIN XX END ELSE BEGIN XX END
```
when it was required for some predicate $P$. The programmer could then plug in the appropriate statements at "XX". Thus the automatic system precluded any syntax errors. A second system was taught to one third of the participating subjects. This programming approach essentially modeled the procedure performed automatically in the first system. A third control group was provided no special tools or instructions for programming in NEST-BE. The measures of programming difficulty used to compare these three approaches to writing NEST-BE programs were: the number of subjects successfully completing a problem, the number of error-free programs per subject, error lifetimes, and number of syntax errors. The experimental results indicate that trivial syntax errors can be reduced in NEST-BE programs by teaching an appropriate procedure. In fact, an additional procedure was developed for JUMP and this learned procedure improved syntactic and semantic error rates in JUMP programs.

As mentioned previously, the extent of control over the variables investigated by the Sime experiments is most valuable, but it would be nice to exercise this type of control in a more realistic programming environment. This was the intention of an experiment performed by J. D. Gannon [GAN76] which is also reported by Gannon and J. J. Horning in [GAN75]. These studies involved two languages, TOPPS and
TOPPSII which were used in a senior undergraduate course in operating systems. The languages were used for four problems assigned during the course. All the students' output was collected in order to determine measures of the number of detectable errors (by the language processor), the number of semantic errors, the average number of runs in which an error goes undetected (computed per problem), and the number of actual errors (not counting repetitions of the same error). The nature of the differences in the two languages rests in the fact that TOPPS contains features similar to those in APL and BLISS while TOPPSII contains more widely used features like those found in Algol. The specific differences comprise the experimental variables. For instance, expression evaluation in TOPPS is performed right to left with no operator precedence and the assignment operator may appear any number of times in a given expression. Expression evaluation in TOPPSII is by the more common operator precedence approach and assignment occurs once in a statement for that purpose.

The experimental results showed that programming difficulty is less for programs written in TOPPSII. But this result must be considered in light of Simé's observation that programmer training may bias results. Gannon and Horning acknowledge that the subjects in their experiment had considerably more experience in Algol-like
languages than in APL-like languages.

In addition, this study demonstrates the problems associated with trying to extend the micro-language experimental methodology. The differences between the two languages must be more tightly controlled. In this study there is surely a relationship between assignment errors and expression evaluation errors—the independent variables should not be so closely related. But this attempt to extend the number of variables which can be tested using the two languages is done for practical reasons—to ask that processors be written for each variation and that experiments be conducted using each such pair of processors is probably unreasonable. The time to manually analyze and classify the errors is alone prohibitive.

Gannon conducted another experiment with H. E. Dunsmore which is reported in [DUN78]. Two actual programming languages were developed so that they differed primarily in features related to data types. The languages were used in a senior level undergraduate course so that, like the previous Gannon experiment, the programming environment was much more realistic than that of the Sime experiments. The measures of programming difficulty in this experiment were similar to those of the prior experiment, with the exception that the only errors considered were those detectable by the
language processor. The researchers note that any experiment which uses data on errors which occur in program runs is implicitly restricting the notion of programming difficulty to the part of the programming process after the first run is submitted. Clearly, the programmer has a considerable investment before the first run, and hence these measures of programming difficulty are in some sense inadequate.

There are two important results of Dunsmore and Gannon's experiment using two languages. The first is that more variable types (as opposed to tokens) indicate lower programming difficulty. In one of the program complexity measures (software effort) evaluated in subsequent Chapters, increased numbers of types yields a larger measure. However, there are other components of software effort and it may well be that increased types would result in a corresponding decrease in these other components. Whether or not this is the case is not determinable from Dunsmore and Gannon's published results. The second result of note is that the total number of decisions did not seem to influence programming difficulty. This result would seem to contradict the pervasive belief that control flow complexity is an important component of program complexity. In order to determine if this seeming contradiction holds up, it would be necessary to determine how the decisions counted by
Dunsmore and Gannon are composed—how many of the decisions are compound conditions made up of more than one predicate. The control flow complexity contention is not contradicted if Dunsmore and Gannon compared \( n \) decisions which are simple predicates with \( n \) decisions which are compound conditions. Here again, the published results preclude determining if the results actually contradict the commonly accepted notion about control flow complexity.

The research reviewed in Section 2.1 has significance for work on program complexity measures because it provides some insight into the measures of programming difficulty which could be used to test proposed measures empirically. These research efforts can also provide insight into currently important factors of language-features and programming methodologies and, in addition, may help indicate how a proposed program complexity measure ought to behave as certain features or methodologies are varied. But the most important implication of these studies for research presented in this dissertation is contained in Shneiderman's point that the development of program complexity measures should be a long range goal of empirical studies. While certain contributions are made by each of these studies to the development of software engineering, it is clear from viewing the entire set of experiments that trying to answer the many questions of language features and programming
methodologies by reliance on controlled experiments alone is totally impractical. What is needed is a measure which can be obtained from actual programs which can help resolve these disputes—a program complexity measure. Such a measure can then be used in a posteriori analyses of existing sets of programs.

Section 2.2. Software Science

This section focuses on research which, as was mentioned previously, has grown out of M. H. Halstead's belief that written material behaves certain fundamental laws and that these laws may be discerned through the approach of natural science. This body of work is now called "software science", although Halstead first labelled his studies "software physics". This initial title, which was used for several years, was also used by other workers to describe an essentially disjoint body of research [KOL75]. Much of the early work in the discipline of software science was concentrated on determining the basic structural properties of computer programs and any laws which govern the basic structure. It seems inevitable that any search for the basic building blocks of computer programs will arrive at the conclusion that programs are composed of operators and operands. Halstead based all his
observations on this assumption. That this conclusion is well-founded is seen in the fact that any program in its machine language version is clearly composed of operators and operands. There is really nothing contained in machine language programs which cannot be classified in this manner. That the choice of operators and operands is extensible to higher level languages is supported by one common approach to natural language semantics. Various researchers in the discipline of linguistics (e.g. [FIL70]) have argued that the meaning of natural language can be represented by concepts (things) and relations among concepts. That is, a natural language representation can be built of operators and operands. Therefore it is not unreasonable to investigate the structure of higher level programming languages from the standpoint of these same entities. (In further support of the contention that natural language is comprised of operators and operands is the research by G. Kulm in which some of the software science relationships were applied to natural language text with very encouraging results [KUL75].) Thus the results in the discipline of software science are motivated by Halstead's basic philosophy and are based on the substantiated belief that operators and operands are the "atoms" of computer programs. In addition, the software science results are consistent with the desire, developed in
Chapter 1, to obtain measure based on the actual source program.

2.2.1. General development

The pioneering observations in software science are contained in [HAL72A]. This one report contains definitions of many of the measures which have been the subject of extensive research interest. These measures are computed per program unit—they are intended as measures of an entire algorithm. The basic software science quantities are:

- \( n_1 \): the number of distinct operators
- \( n_2 \): the number of distinct operands
- \( N_1 \): the total number of operator occurrences
- \( N_2 \): the total number of operand occurrences
- \( n = n_1 + n_2 \): the number of types in the program
- \( N = N_1 + N_2 \): program length (roughly the number of program tokens)

This first work by Halstead also contains the first software science relationship. He conjectured that program length can be approximated given simply the number of types in the program. In other words, Halstead's intuition indicated there ought to be a relationship between \( N \) and \( n \).
The relationship he investigated has become known as the length equation and is expressed:

\[ N' = n_1 \log n_1 + n_2 \log n_2 \approx N \]  

(2)

The length equation has been submitted to a considerable amount of empirical verification. The first support was provided by Halstead in his original software science paper. The first 14 algorithms from *The Communications of the ACM* were analyzed and the average percentage difference of values for \( N' \) and \( N \) in these programs was found to be \(+3.1\%\). The worst \( N' \) estimate was 33.7\% over the actual \( N \) value, certainly not an unreasonable worst case. Three other studies of the length equation provided similar results on considerably larger data sets. J. L. Elshoff tested the length equation on 154 PL/I programs [ELS76] and M. Magidin and L. Viso performed the same analysis on 50 algorithms from *The Communications of the ACM* [MAG76]. Finally, N. Bulut, Halstead, and R. Bayer compared \( N' \) to \( N \) for 429 FORTRAN programs from a program library at Purdue University [BUL74B]. These studies all support the

(2) Throughout this work "log" will be used to denote a logarithm to the base 2.
contention that the behavior of $N'$ over a large number of programs approximates $N$ very closely.

However, while the length equation seems to perform well on the average in empirical studies, an analytical explanation of $N'$ has proved elusive. J. C. Ingojo showed that, in at least one instance, the length equation holds over entire large programs, and also for the individual modules which make up the program [ING75]. Thus it would seem that the length equation captures something significant about the type-length relationship for logical units, and thus may be capturing a basic "law" of the underlying thought processes [HAL77]. Halstead and R. Bayer attempted an analytical explanation of $N'$ by developing models for program generation [HAL72B]. The programs produced by one model are quite close to actual programs, but the model is not amenable to reasonable analysis. Therefore the three other models were developed. The first is quite simplistic and the expected length of programs generated by this model is analytically derived. A second model generates slightly more realistic programs and the expected length of these programs is obtained by easily solved recursive relations. The third model produces programs whose expected lengths are determined by Monte Carlo simulation. In all three cases the expected lengths agree quite closely with those predicted by $N'$. Thus Halstead and Bayer's work in this
area lends considerable support to credence in $N'$, but does not actually provide an analytical justification of this length relationship. However, following Halstead's natural science philosophy, one must acknowledge that simply because he or she cannot explain a particular relationship does not in itself detract from the relationship's accuracy.

Another measure of interest defined in Halstead's early report is an attempt to quantify program size in terms of bits. Given $n$ entities, the minimum number of bits required to represent those $n$ entities would be $\log n$. (In real terms it would be the smallest integer greater than $\log n$, but the interest is in a theoretical quantification.) Therefore, if the $n$ entities occur $N$ times then the number of bits required to express those occurrences would be $N \log n$. Halstead called this measure program volume:

$$V = N \log n$$

It is also worth noting that if the $n$ entities have equal probability of occurrence, then the software volume measure $V$ is exactly C. Shannon and W. Weaver's measure of message information content [SHA49].
Once this measure of volume is obtained, the obvious question is whether or not the measure has a smallest possible value for a given algorithm. In this presentation the symbol "*" is used to denote such minimum quantities, thus the question concerns the existence of a quantity \( V^* \). Clearly, such a theoretical value would be language independent, i.e., \( V^* \) should be constant regardless of the particular language chosen to express the algorithm. This invariance of a measure \( V^* \) is in fact the second significant relationship presented in [HAL72A]. In this early development \( V^* \) was labelled "internal quality" (10) and an expression was developed and tested for this measure. However, for the moment it is more instructive to simply let \( V^* \) represent theoretical minimum volume and consider measures which may be based on \( V^* \).

One possibility for the use of \( V^* \) is an expression to quantify how closely an actual program resembles the ideal solution to the problem that the actual program solves, in terms of volume as measured in bits. This approach to defining a measure of the level at which a program is written is presented in Halstead's summary of software science research *Elements of Software Science* [HAL77]. A simple expression which captures the desired notion is:

\[
L = \frac{V^*}{V} \quad \text{level}
\]
Thus if an algorithm is expressed ideally (in terms of bits) it would have $L = 1$. In all practical cases, $L < 1$. An immediate consequence of this intuitively appealing quantification of the level at which an algorithm is expressed is that $L \times V = V^*$, and is therefore constant by definition.

But, as mentioned above, Halstead submitted the invariance of $V^*$ to empirical test, which would seem quite unnecessary given the definition of $L$. In this earlier work, however, a different expression for program level is presented, and it is this measure, which is denoted $L'$, that was used to test the invariance of the product of level and volume. The expression for $L'$ is:

$$L' = \frac{2 \times n_2}{n_1 \times N_2}$$

The test presented in the early paper involved five versions of the same algorithm expressed in different languages. In this case all the calculated values of $L' \times V$ are within 30% of their mean. Halstead acknowledged that $L'$ and IQ might require refinements in their development, but this preliminary evaluation provided encouragement for the invariance premise. The later presentation using $V^*$ is the result of the ensuing development.
Thus the underlying hypothesis tested in Halstead's preliminary study is the extent to which the product $L' \times V$ is constant for a given algorithm—in other words, the extent to which the behavior of $L'$ in the product $L' \times V$ resembles the defined behavior of $L$ in the product $L \times V$. (It turns out that this behavior of $L'$, relative to the defined level measure $L$, is quite important in evaluations of the software effort measure which receives considerable attention in later chapters.) N. Bulut also investigated this invariance relationship involving $L'$ using the first 14 algorithms from *The Communications of the ACM* [BUL73]. For these algorithms $L' \times V$ was calculated for both ALGOL versions and machine-code versions (machine code being that for a CDC 6400). It was determined that a strong positive correlation exists between the $L' \times V$ products for the versions in the two languages. These two studies indicate that $L'$ approximates the defined behavior of $L$ in the product (level measure) times volume.

Further study of some of the cases in which this level-volume product did not remain relatively constant for implementations of the same algorithm in different languages led to the observation that certain undesirable program constructs seem to perturb the expected invariance. A first attempt to identify these constructs appears in [HAL73A]. In this study a trivial problem, squaring the sum of the
input parameters, was solved by 18 different programs. The value of \( L' \times V \) was determined for each of the programs. In those cases where the product is considerably aberrant, the following program constructs can be observed:

1. self cancelling operations.
2. synonymous operands.
3. ambiguous operands. One operand used for two different purposes.
4. unfactored expressions.

The removal of these "impurities" improves the invariance of \( L' \times V \) in the 18 resulting programs.

However this result for \( L' \times V \) might be expected simply because of the trivial nature of the problem. Therefore N. Bulut conducted a more extensive investigation of the effect of the impurities on the invariance relation \([BUL73]\). Bulut and Halstead reported the following two additional impurity classes in \([BUL74A]\):

5. common subexpressions.
6. unnecessary assignment to temporary variable.

The basis of Bulut's study of \( L' \times V \) was the ALGOL and machine code versions of the 14 algorithms from The Communications of the ACM mentioned previously. Actually two sets of machine code versions were produced by two different FORTRAN compilers. Also hand generated machine code versions were produced for seven of the algorithms.
When the $L^* \times V$ values were computed and compared, the handwritten versions, which contained fewer impurities, seem to help preserve the invariance when compared with the ALGOL versions. This same effect was noted when PILOT and ALGOL and PL/I and ALGOL versions were analyzed. Thus the negative effect of impurities on the invariance of $L^* \times V$ is established. Bulut also showed some negative effect of impurities on the extent to which $N^*$ predicts $N$.

Now returning to the discussion of the uses of $V^*$, it does seem that using an abstract notion of minimum volume to quantify program level does have intuitive appeal, but to be of any practical value there must be a way to quantify $V^*$. Halstead proposes a quantification in [HAL72A]. If for a particular problem one has available a built-in function which solves exactly that problem, then an invocation of the function is a program which solves that problem. In such cases two operators are required—the function name and an argument list delineating operator. Therefore the number of arguments to the function becomes important and each operator or operand will only occur once. This discussion motivates the following development of a quantification of $V^*$:

$$V^* = (N1^* + N2^*) \log (n1^* + n2^*)$$
$$= (n1^* + n2^*) \log (n1^* + n2^*)$$
\[(2 + n2^*) \log (2 + n2^*)\]

The extent to which this quantification of \(V^*\) is adequate is difficult to establish, but it does not seem unreasonable. As with the \(L^*\) measure, this quantification of minimum volume becomes pertinent in the evaluations of the software science measure \(E\), which is described shortly.

The software science measures presented so far in this section deal with properties of the structural aspect of programs— they quantify notions which pertain primarily to the program itself. But Halstead’s philosophy of applying a natural science approach to the study of programming motivates an attempt to develop a measure which quantifies some aspect of the programmer-program relationship. Halstead based his investigation on the conjecture that "Since algorithms are in some sense distillations of thought, their fundamental nature may well be related to mental effort, as heat is related to energy or to physical work." [HAL72A]. He noted that this task was complicated by the fact that there were no adequate measures of "heat" in programs. The effort measure \(E\) is Halstead’s attempt to define a heat analogy for programs. In attempting to develop the measure using the structural properties which had already been identified, it was reasonable to require
the following characteristics of the measure:

1. E ought to be directly proportional to the volume of the program.
2. E ought to be inversely proportional to the level of the program.

With these two requirements, and again adopting the natural scientist's approach by seeking the simplest relationship with the desirable properties, the following quantification of $E$ results:

$$E = \frac{V}{L}$$

so that

$$E = \frac{(V + 2)}{V^*} \quad (3)$$

This reasoning in the derivation of $E$ first appeared in [HAL72C]. However the measure derived and tested in this report is defined using $L^*$ instead of $L$. This quantification of effort is expressed as

$$E' = \frac{V}{L^*}$$

---

(3) The symbol "\(\star\)" will be used to denote exponentiation.
In order to investigate this measure Halstead chose what is not an unreasonable measure of programming difficulty, actual programming time, as a basis for the evaluation. However, in light of the empirical studies reviewed in Section 2.1, it is clear that this is but one of many possible ways to quantify programming difficulty. In addition, like the other measures which have been used in empirical studies, this is not a complete measure—it does not fully capture programming difficulty, even when difficulty is simply considered from the standpoint of program generation. Furthermore, as pointed out in Chapter 1, a program complexity measure cannot be expected to capture fully a measure of programming difficulty resulting from the generation activity because generation involves "backing up and starting over" which is obviously not reflected in the resulting code.

But even with these limitations of the approach he used, Halstead found his evaluation encouraging. Six machine language programs were produced, keeping track of the time required to write each program. Then the value of E for each program was computed. Halstead noted that the number of hours to produce the programs varies by a factor of 96, while the ratio of hours to E varies by only a factor of 4.4, with an average of 65,800. This led Halstead to the possibility that E might do more than simply correlate well
with programming time. J. M. Stroud suggested that psychological time has a "fine structure" composed of "moments" [STR66]. Moments can be compared to computer processor speeds—the human mind is limited in the number of elementary mental discriminations it can make per unit time. Stroud established a moments per second range of between 5 and 20. Halstead divided the average ratio of hours to $E$ by the number of seconds per hour and got 18. This falls within Stroud's range and lends some credence to the claim that $E$ measures the number of elementary mental discriminations required to produce a program. This is an extremely powerful result and considerable research has been devoted to investigating its validity. Several important empirical studies of $E$ are presented in considerable detail in 2.2.3.

However there are a few additional research efforts which should be mentioned in a general introduction to software science research. A. Elci used the assumption that each unit increase in resources doubles the number of types needed in the operating system which handles the resources, along with a form of the length equation, to derive an estimator for the number of machine language instructions in an operating system which handles the resources [ELC75]. Thus, if $RA$ denotes the number of system resources, and $P$ the number of instructions, Elci tested empirically an
equation of the form

\[ P = f(RA) \]

Using data from 14 operating systems, Elci found a correlation between his estimator and actual operating system size of 0.954. Elci's results lend some support to the possibility that some of the software science relationships may actually be probing basic structural properties of products of man's thought processes.

S. H. Zweben extended the understanding of the basic structure of computer software in his studies of the distributions of operators and operands in programs which are reported in [ZWE77]. His investigations revealed that Zipf's Law as modified by Mandelbrot provides an adequate model of operator occurrence frequencies in programs. (This operator frequency distribution model is later modified in [ZWE79].) However, and possibly somewhat surprisingly (given that Zipf's Law applies to frequency distributions of occurrences of all words in natural language text), Zipf's Law does not provide an adequate model of operand frequency distributions. Zweben used one of Bayer's models of program generation to derive the following expression for the number of occurrences of the ith most frequently occurring operand:
\[ f_2(i) = f_2(1) \exp(n_2 - 1) / (n_2 - 1) \]

These models of operator and operand frequency distributions add to the confidence that these entities do in fact form the basic building blocks of computer programs. If these entities were not the "atoms" of computer programs, then the regularity of their occurrences would seem far less likely.

The following two parts of this section on software science research are particularly relevant to the research results presented later and are therefore presented separately. But the works presented in 2.2.1 are important for obtaining an accurate impression of the discipline which has produced these other works. Software science has produced several results which have increased our knowledge of software and software development and M. H. Halstead, with his innovative synthesis of natural science and the study of software, deserves a tremendous amount of credit from those of us who have made use of his ideas in our own work and from the computer science community as a whole.

2.2.2 The Counting Strategy

The preceding general development of software science research taken as a whole supports the notion that operators and operands are the basic building blocks of computer
software. And, perhaps because of its intuitive appeal, the classification and identification of a language's types into these groups is not something that has been of paramount concern in most of the software science research efforts. The notes by Zweben in [ZWE73] are the first acknowledgment of the potential problem. In this report Zweben points out that some tokens always occur in pairs and that some of these pairs are linked semantically. The IF...THEN alternative construct is a case in point: it would make little sense to classify THEN as a distinct operator. Another example is the "," in the language PILOT which always occurs with the storage operator "++" and serves as a statement separator. However the tokens "," and "++" have meaning by themselves. A small test indicated that the length equation works better if the comma in PILOT is counted as a distinct operator. This suggests that using semantic information to classify operators may be warranted.

J. L. Elshoff showed that the way operators and operands are counted can have a significant effect on some of the measures in [ELS78A]. In this study eight variations of the PL/I counting strategy described by Bulut [BUL73] were investigated. This basic strategy, described by Elshoff, is representative of the counting strategies used in many of the early works and characterizes much of the
approach to the counting strategy used later in this dissertation. Therefore a summary of this strategy is provided:

1. Operands. Specific constants and simple variables are operand occurrences. Any reference to an array is counted as an occurrence of an operand corresponding to the array name—each array element is not a distinct operand. Similarly, major structure names, minor structure names, and elementary item names in PL/I structures are considered to be distinct operands.

2. Operators. The following PL/I tokens are distinct operators:

```
+      -      *      /      <
==     ||     <    >    <=
>=     !=     >    <    &
|      SIGNAL ALLOCATE LOCATE DELAY
FREE FETCH REVERT WRITE
READ OPEN CLOSE PUT
DISPLAY WRITE GET INITIAL
\ (qualified name separator)
```

Other tokens and token pairs are handled as follows:

a. Grouping operators. The token pairs (...,) (in expressions, around argument lists, and in INITIAL options), DO...END, and BEGIN...END are counted as a single grouping operator.
b. WHILE and TO...BY. The WHILE in a repetitive DO statement and the TO...BY in an iterative DO statement are counted as distinct operators. Note that the DO...END portion of these loops are counted as in a.

c. Array subscripts. The parentheses around array indices are counted as distinct operators from those parentheses which are grouping operators.

d. GO TO. Each explicit transfer of control GO TO <label> is counted as a distinct operator for each unique instantiation of <label>. It should be noted that labels on statements are not counted as operands.

e. Subprogram invocations. Each invocation CALL <entry name> or <function name> is counted as a unique operator for each distinct instantiation of <entry name> or <function name>.

f. IF statements. Occurrences of IF...THEN and IF...THEN...ELSE are counted as occurrences of one unique alternative operator.

g. Statement terminators. Semicolons are counted when they occur on statements in which another token has been counted. Thus semicolons on PROCEDURE statements and declarations are not
counted.
The counting strategy presented above differs from that used in Bulut's research only in that I/O operators are counted. It was originally felt that, since the software science measures are believed to hold most accurately for "pure algorithms", the I/O present in an algorithm's implementation in an actual program is superfluous to the algorithm itself. This assumption bears up quite well for the type of algorithms investigated in these early studies as they are primarily numerical computations which require little I/O. In fact, many are expressed as subroutines with no I/O. However, the application of these measures to production programs probably necessitates the relaxation of this approach. Who would want to contend that I/O contributes nothing to the effort involved in writing production COBOL programs?

Using this counting strategy as a basis, Elsholz varied certain aspects of the strategy in basic measure counts of 34 PL/I programs. The eight variations used in the study reflected attempts to maximize unique operator counts and attempts to minimize unique operator counts. The results of these varying strategies indicate that while some of the measures are relatively insensitive to counting strategy variations (for example program volume), others are affected considerably. The percentage difference between the E*
measure for a neutral strategy and that for a minimizing strategy is \(-34\%\), indicating that the measure \(E'\) is somewhat affected. But it is worth noting that the effect on \(E'\) must be primarily the result of a corresponding positive percentage difference on \(L'\), since \(V\) remains relatively constant.

Thus it would seem that the choice of a counting strategy can affect experimental results and that determining a well justified counting scheme remains an unsolved problem in software science research. This is not to say that most of the decisions are not straightforward, but rather that there are still a few issues which should be resolved by a generally accepted schema. In the analysis of software effort in subsequent chapters, the counting strategy presented in detail earlier in this subsection is applied. It is the same strategy used by Elshoff in his experiments which averaged the extremes of unique operator occurrences. In Chapter 3, the analysis of \(E\) leads to an analytical justification of a resolution to one counting strategy issue and may demonstrate one possible approach to aid in the development of the desired classification schema.
2.2.3 Evaluations of $E$

The empirical research which has been performed to date to evaluate $E$ has been most encouraging. These experiments taken as a whole lend considerable support to the contention that $E$ is an adequate program complexity measure. Numerous studies which indicate a positive correlation between $E$ and several different measures of programming difficulty are reviewed. Within these studies, however, there are two principal points on which the experiments show variation:

1. Whether or not impurities in the programs used in the experiments should be accounted for in calculating $E$; and if they should, how?
2. What quantification of level should be used to obtain an effort-like measure?

Two of the early empirical evaluations exemplify these issues and so are reviewed in detail.

The first attempt to investigate $E$, after Halstead's original experiment which was described in the general development of software science, is reported by Halstead and P. M. Zisli in [HAL73B]. In his experiment Zisli [ZIS73] developed a formula for estimating actual programming times from the effort measure and compared the formula's estimations with actual programming times. The implementation times were obtained as follows: 12
algorithms from *The Communications of the ACM* were selected and specifications were written for each problem. The amount of time required to produce a working program from the specification was recorded for each program written. Each of the algorithms was coded in a language randomly selected from FORTRAN, PLI, or APL.

In this same study Zislis also investigated the invariance of the product of a level measure and \( V \), and this investigation led to one approach to accounting for program impurities in the effort measure. The measure of level used by Zislis was

\[
L'' = \frac{(4 \times n^2)}{(n_1 \times N)}
\]

and in analyzing \( L'' \times V \) he found that the invariance was present on the average but that individual instances varied significantly. Zislis hypothesized that this might be due to impurities in the programs and therefore tried to account for these impurities by obtaining a purified value of \( n^2 \), which he called \( n^2' \). This purified \( n^2' \) was obtained in the following manner: the product \( L'' \times V \) is expressible as

\[
L'' \times V = \frac{(4 \times n^2 \times \log n)}{n_1}
\]

The work on impurities by Halstead [HAL73A] and by Bulut
[BUL74A] supports the hypothesis that the measures $n_2$, $N_1$, and $N_2$ are affected by the presence of impurities, but that $n_1$ is not. Therefore an $n_2''$ value which satisfies

$$n_2'' = (n_1 \times V^*) / (4 \log (n_1 + n_2'))$$

will make $L'' \times V$ invariant. If the lack of invariance was the result of impurities, then the $n_2''$ value would reflect the number of distinct operands in a purified version of the program. It is somewhat unfortunate that this cause of invariance is not at least supported by a limited number of examples in the Zislis report, but it is not. This is not to say that the approach might not be warranted, but only that it requires further substantiation on the data to which it is applied.

Zislis uses this adjusted number of operator types in two places in his effort formula. It is used in a modified volume formula

$$V' = N'' \log (n_1 + n_2'')$$

where

$$N'' = n_1 \log n_1 + n_2' \log n_2'$$
This modified volume formula is then selectively inserted into the definition of $E$ to obtain

$$E(ZIS) = \frac{(V \times V')}{V^*}$$

and $E(ZIS)$ is used in the timing predictor

$$T(ZIS) = \frac{E(ZIS)}{S}$$

where $S$ is the Stroud number mentioned earlier (approximately equal to 18). Thus the Zislis timing equation was developed to reflect the influence of impurities on volume by adjusting $n2$ in the original effort definition. Halstead [HAL75] observed later that the correlation coefficient of values for $T(ZIS)$ and actual programming times is .92 and that $T(ZIS)$ is, in fact, not an inadequate predictor of actual programming times. Later, as reported in [HAL77], Halstead compared $T(ZIS)$ with a simple measure of the number of statements in the program. Halstead used the first program written for each problem specification and found that the correlation of the number of statements and programming time was only .694. Since number of statements correlates less well with programming time than does $E$, this bolsters the confidence in $E$ as a program complexity measure.
R. D. Gordon conducted another experiment using programming times as measures of programming difficulty to evaluate the extent to which the notion of software effort satisfies the criteria of a program complexity measure [GOR76]. Eleven problems were randomly selected from two sources: Knuth's *The Art of Computer Programming* [KNU68] and *A Collection of Programming Problems and Techniques* by H. A. Mauer and M. R. Williams [MAU72]. The problems were specified informally in these sources. Gordon wrote programs to solve each of the 11 problems, working on only one problem in a given day. The time required to produce a tested program for each problem was carefully logged. After each program was completed, a careful count of the basic software measures was obtained.

Included in Gordon's report of this experiment is a first attempt to explain analytically why a software effort measure ought to yield actual programming times. Gordon presented the following nine steps in the development of his effort formula and timing equation:

1. A program consists of $N$ selections from $n$ elements.
2. A binary search of $n$ elements requires $\log n$ comparisons.
3. A program is generated by making $N \log n$ comparisons.
4. Therefore $V = N \log n$ is a count of the number of
comparisons required.
5. The number of elementary mental discriminations required to complete one comparison measures the difficulty of the task.
6. The software measure $L$ is the reciprocal of the difficulty.
7. Therefore, $E$ is the count of elementary mental discriminations required to generate a program and $E$ is given by:

$$E = \frac{V}{L}$$

8. $S$, the speed with which the brain makes elementary mental discriminations can be obtained from psychology [ISTR66] as:

$$5 \leq S \leq 20 \quad \text{(discriminations per second)}$$

9. Therefore the time to generate a preconceived program, by a concentrating programmer, fluent in the language being used, is:

$$T(GOR) = \frac{V}{L \cdot S}$$

(The concentrating programmer criterion supports the
use of an $S$ value near the upper end of the range of mental discriminations per second. The fluency requirement insures that the programmer works on the problem at hand and not her or his language deficiencies.

Again it must be emphasized that any deficiencies in an analytical justification for a measure's behavior does not in itself detract from the predictive abilities of that measure. With this point of view in mind, a few comments about these nine steps are appropriate:

Steps 1-3. These steps imply that programs are written by "preconceiving" the set of operator and operand types which will appear in the program and then choosing from these types when the program is written. This may reasonably accurately model the process of program generation, but programs are written by choosing from the set of operators afforded by the implementation language, plus any operators (subprograms) the programmer defines himself. The set of operands is infinite. Step 2 implies that mental selection from among $n$ types behaves, in some quantitative sense, as a binary search. Again, whether or not this is the case is at least a conjecture which requires additional substantiation.
Step 4. This step does in fact follow, given the premises of steps 1-3.

Steps 5-6. This is the weakest part of the development. It has been shown that given two implementations of the same algorithm, one in a low level language like assembler and the other in a higher level language, that the software science level measure \( L \) must be lower for the low level language implementation. Any of the experiments which substantiate the invariance of the product of level and volume must support this claim, since volume for the low level language version will obviously be greater.

This notion of level is defined as a quantification of how concise the program statement actually is (in volume) as opposed to some (theoretical) minimal expression. In this sense level is indeed the reciprocal of difficulty—"difficulty" may be defined as the reciprocal of software level. However, there is considerable question as to whether the "difficulty" in step 5 is the same as this software "difficulty" of step 6. The difficulty in step 5 seems to refer to the number of mental discriminations required to make one comparison (in selecting a token from the \( n \) possible types). Consider this comparison process in a low level language. The number of mental discriminations
should be small because the types being considered are not particularly abstract. In a high level language the number of discriminations per comparison should be larger. Thus the difficulty referred to in step 5 increases as the implementation language uses more abstract operators and operands. This is clearly not the notion of software science difficulty of step 6, and $L$ does not measure the number of elementary discriminations per comparison.

Steps 7-9. These steps do follow if the premises in steps 1-6 are sound.

In summary there are at least conjectures in steps 1-3 which require some type of additional substantiation and it may well be that the deficiencies enumerated for steps 5-6 invalidate this analytical development for the Gordon timing equation.

However the experimental results uphold the accuracy of the equation. In the actual timing equation used, Gordon accounted for impurities by using $N'$ for $N$. Gordon's timing equation is thus expressed as

$$T'(GOR) = \frac{(n_1 \times N_2 \times N' \log n)}{(2160 \times n_2)}$$

where $S = 18$ which yields 1080 mental discriminations per second. The use of $N'$ in $T'(GOR)$ presents a problem similar
to that of Zislis' use of $n2^2$ in that the use of $N'$ to account for impurities is not substantiated by other software science research. What is well established is that the length equation $N'$ more closely approximates $N$ if the program being considered has no impurities. This does not directly imply that if impurities are removed from a program with length $N$ that the resulting program will have length equal to the $N'$ value of the original program. It is this latter assumption which Gordon makes use of.

Despite any drawbacks in the analytical development of $T'(GOR)$, this measure is a reasonably accurate estimator of programming times. Gordon found that the mean value for $T'(GOR)$ was 34.1 and the mean of actual programming time for the 11 programs was 35.0. Two cases showed rather significant errors, but clearly the experiment improves confidence in the measure. Halstead notes [HAL77] that the correlation of $T'(GOR)$ with actual programming times in this experiment is .93 giving additional support to Gordon's positive conclusions concerning the measure.

The experiments by Zislis and Gordon provided the impetus for additional empirical evaluations of $E$. Y. Funami and Halstead reported in [FUN76] a high positive correlation between number of errors which occur during program generation and $E' = V / L'$. Akiyama published the
number of errors which occurred in the development of each of nine modules [AKI72] and reported correlations between number of errors and number of program steps of .83 and between number of errors and the sum of the number of decisions plus calls of .92. Funami and Halstead had to do some estimating to get the basic software science measures from Akiyama's published information. Using these estimated values of n1, n2, N1, and N2 they found a correlation between E' and number of errors of .982. Using the timing equation T(GOR), Funami and Halstead estimated the total time to generate the nine modules at 84 man-months, which agrees reasonably well with Akiyama's published value of 100 man-months.

Another attempt to substantiate software effort using a variety of measures of programming difficulty is summarized by A. Fitzsimmons and T. Love in [FIT78]. In the first two of their tests of software effort the authors used data published by T. Love and A. B. Bowman in [LOV76]. This report by Love and Bowman contains the data on debugging time and error counts obtained originally by Gould in his empirical study [GOU74] and data from Weissman's empirical study of program understanding [WEI74], along with the basic software science measures for the programs involved in the two sets of experiments. The Gould data comes from his experiment on planted bugs in programs. Fitzsimmons and
Love found a positive correlation of .78 between the number of wrong guesses made by programmers before the bug was correctly identified and $E = \frac{V}{L}$. These authors conjectured that this correlation would be improved by considering those subjects who took longer to find the error. They felt that taking longer to find the error would require a more thorough understanding of the entire program, and it is this total understanding that $E$ actually measures.

The data used by Fitzsimmons and Love from the Weissman study were test scores from the quizzes administered after an initial program study period and after additional study time, as well as the rankings of understanding from self evaluations. The authors found that $E$ is not significantly correlated with first quiz scores or first self evaluations. They did note a high negative correlation between $E$ and the second self evaluations, and a stronger negative correlation between $E$ and second quiz scores than that observed for the first quiz scores. That the correlations move in the anticipated direction after additional study lends some support to $E$ as a program complexity measure.
In a third evaluation of E, Fitzsimmons and Love conducted their own experiment to evaluate E in light of accepted notions of control flow complexity and complexity incurred by poor choice of program identifiers. The authors wrote nine programs to solve the same problem, varying control flow and mnemonic complexity. Experimenting using percentage of lines correctly recalled as measures of programming difficulty, the authors concluded that control flow complexity variation is the only significant factor of programming difficulty in the nine programs. They also found a strong negative correlation, $-0.81$, between E and the percentage of correctly recalled lines. In fact, E is a considerably better indicator than number of program statements as the correlation of number of statements with percentage of correctly recalled statements was only $-0.70$.

Finally, Fitzsimmons and Love report a strong correlation between E and "delivered bugs". They used data from three large G. E. software projects which included records of the number of bugs located after coding and initial testing. The data also permitted the estimation of the basic software science measures. The authors found a positive correlation of $0.76$ between E and numbers of delivered bugs in the three systems. This study alone, and certainly the four studies reported by Fitzsimmons and Love, support E as a program complexity measure.
The most recent empirical study of software effort which has been described in the published literature was conducted by R. Gordon. In [GOR79A] Gordon used 46 pairs of programs which are supported by at least one author as representing good and bad versions of the same program. These pairs of programs serve as the indicators of programming difficulty. The pairs were obtained from [CHU76], [GIL74], [KER74], [KNU74], and [NIR74]. Gordon calculated $E' = V / L'$ for each of the 46 pairs. His choice of this measure of effort is significant. In his previous empirical study of effort using actual programming time as the indicator of programming difficulty, Gordon adjusted his effort measure to account for the influence of impurities. The impurities were felt to be significant factors which should be accounted for when software effort is used to predict programming difficulty incurred in program generation. However, in program understanding any impurities would influence difficulty in understanding the program and so no attempt is made to try to adjust for this influence in the measure used in this experiment.

An attempt was made to be fair in determining when the difference in an $E'$ value of a poor version of a program and the $E'$ value of the corresponding improved version is significant. To this end Gordon determined that the difference must represent a change of ten percent from the
values of the poor version. Using this criterion Gordon found that in 40 of the 46 cases the measure \( E' \) agreed with the published contentions concerning these pairs. \( E' \) performed considerably better than an alternative measure, the number of executable program statements, as this measure agreed with the original contentions in but 31 of the pairs. The measure also works better than \( V \) as a program complexity measure, based on this data. This would indicate that the effort measure \( E' \) works better than the measure \( E \), the original defined measure, at least for this data set. However, the actual difference in performance of \( E' \) and \( E \) of three program pairs may not be significant. It is certainly not as great as the difference between the number of executable statements and \( E \).

In a separate evaluation of software effort, Gordon determined the effect on \( E' \) of removing impurities from programs. In [GOR79B] he reports on the effect on \( E' \) of removing impurities, from the six impurity classes which are enumerated in Section 2.2.1. One of the impurity classes analyzed by Gordon is also considered in Chapter 3 and hence is of particular interest. In his analysis of the effect on \( E' \) of assigning the value of a common subexpression, which is repeated once, to a temporary variable and using the temporary in place of the expression occurrences, Gordon found that the expression had to be of length greater than
five. Otherwise the expression would have to be repeated more than once. This conclusion supports a result presented in Chapter 3. Similar results were observed for the other impurity classes.

In summary, software science research has been based on the philosophy that programs and programming can be studied using a natural science approach. Applying this philosophy, Halstead used the basic assumption that programs are composed of operators and operands to define certain basic measures. These measures have been used in a number of relationships which can be viewed as principles of the structure of programs. In addition, software science has spawned a program complexity measure $E$ which has received considerable empirical substantiation.

The software effort measure has several variations and has been used in slightly different timing equations. These varying forms are summarized below:

$$E = \frac{V}{L} = \frac{V + 2}{V^*}$$
$$E' = \frac{V}{L'} = \frac{V}{(n \times n2) / (n1 \times N2)}$$
$$E(ZIS) = \frac{(V \times V')}{V^*} \quad \text{(where $V'$ is developed to account for program impurities)}$$
$$T(ZIS) = \frac{E(ZIS)}{S} \quad \text{(where $S = \text{Stroud number 18}$)}$$
\[ T' = \frac{E'}{S} \]
\[ T'(GOR) = \frac{(N' \log n)}{(1080 \times L')} \]  
(where \( N' \) is the length equation)

\[ T = \frac{E}{S} \]

All of the evaluations of software effort which have been performed since the early experiments by Zislis and Gordon have used \( E \) or \( E' \), or the corresponding timing equation \( T \) or \( T' \). Both \( E \) and \( E' \) have received considerable support as program complexity measures. It is worth noting that the difference in the two measures rests in their different quantifications of program level. It has been pointed out that \( L' \) can result in anomalous behavior [OLD77] and there has been some discrepancy in the particular expression used for \( L' \) (e.g. [ZIS73]). Therefore, since there is evidence that \( L \) is the more consistent measure of level, it is used in the effort measure \( E \) which is evaluated in the subsequent chapters. It would seem that if there is a weakness in the measure \( E \), it might be in the quantification of minimum volume, \( V^* \). However, refinement of this measure is outside the scope of this particular research effort.
Section 2.3 Control Flow Complexity Measures

Control flow complexity is an important aspect in program complexity. That this aspect of program complexity is significant is supported by the number of recent attempts to quantify control flow complexity. It is also given considerable significance by the structured programming controversies which seem to focus attention on choices of program control flow constructs. Definitions and examples are presented in this section for several approaches to quantifying control flow complexity which have appeared in the open literature. Since these measures have appeared recently they do not have the volume of development and evaluation of the software effort measure E, and so their presentation is more succinct. New analyses of these measures are contained in Chapter 4.

2.3.1. Program Knots

M. R. Woodward, M. A. Hennell, and D. Hedley defined a measure of control flow complexity in [WO079]. The measure is based on control flow arcs associated with the actual source program. Essentially what is done is to draw arcs from every explicit transfer of control to the destination of that transfer. Figure 3 provides a simple example in which a FORTRAN implementation of an IF...THEN...ELSE
construct contains one knot.

```
IF (...) GO TO 10
   .
   .
GO TO 20
10  
   .
   .
20  CONTINUE
```

Program Segment Containing One Knot

Figure 3

This notion of arc intersections is formalized in the following definition:

**Definition**: Let an explicit transfer of control in a program $P$ from point $a$ in the source code to point $b$ in the source code be denoted by $(a, b)$. Then an explicit transfer of control $(p, q)$ gives rise to a knot with respect to transfer of control $(a, b)$ if either:

1. $\min (a, b) < \min (p, q) < \max (a, b)$ and $\max (p, q) > \max (a, b)$

or
2. \( \min (a, b) < \max (p, q) < \max (a, b) \) and \( \min (p, q) < \min (a, b) \)

2.3.2. Normal Number

This measure of control flow complexity is based on normal forms of flowgraphs (4). Informally, the normal form of a flowgraph \( G \) is a corresponding flowgraph \( N_G \) where for the same input the sequence of steps in executing \( N_G \) is exactly the same as the sequence of steps executed in \( G \). (Flowgraphs which are equivalent in this manner are said to be executionally equivalent.) In addition, \( N_G \) is an "almost tree" in that there exists exactly one simple path from the entry node to any other node. This gives \( N_G \) the property that any back arc from a node \( A \) in the flowgraph points back to a node which lies on a branch which contains \( A \). That an almost tree has this property is demonstrated in [CUL77].

The flowgraph which appears as Figure 4 has the normal form of the flowgraph in Figure 5. The normal form in Figure 5

---

(4) **Definition:** A flowgraph is a directed graph in which the nodes represent program predicates or other program actions and the arcs represent program control flow. In addition, the directed graph contains a single node with in-degree zero, called the entry node, and a single node with out-degree zero, called the exit node. Every node in the directed graph lies on some path from the entry node and there is some path from every node to the exit node.
is produced simply by producing multiple copies of nodes in the flowgraph of Figure 4. (Note that the replication of the exit node is irrelevant.) This process is called node splitting, and receives additional attention in Chapter 4.

A formalization of the technique for deriving the normal form of an arbitrary flowgraph can be found in [CUL77].

The normal form of a flowgraph is used in a control flow complexity measure in the following manner:

Definition: The normal number, NN, of a flowgraph G is the (minimal) number of simple loops in the normal form flowgraph of G.

2.3.3. Cyclomatic Number

T. J. McCabe in [MCC76] defines another control flow complexity measure which is also based on the flowgraph representation of a program. This measure relies heavily on two results borrowed from graph theory. The work by C. Berge [BER73] contains the following definition:

Definition: The cyclomatic number $C'(G)$ of a graph $G$ with $v$ nodes, $e$ edges, and $p$ strongly connected components is
Normal Version of Flowgraph in Figure 4

Figure 5
This definition leads to the following theorem (also from Berge):

**Theorem:** In a strongly connected graph $G$ the cyclomatic number $C(G)$ is equal to the maximum number of linearly independent circuits in $G$.

"Linearly independent circuits" may be interpreted as follows: In $G$ assign a unique number to each edge of the flowgraph. Then each circuit in $G$ which does not traverse the same arc twice (i.e., elementary circuits) can be represented by a vector of ones and zeros. The $i$th element of the vector is a one if the $i$th edge of the flowgraph appears in the circuit, and zero otherwise. Circuits which are not elementary are represented by letting the $i$th element of the vector corresponding to a particular circuit be $j$ if the $i$th edge of the flowgraph is traversed $j$ times in the circuit. Thus "linearly independent" refers to the set of vectors corresponding to the circuits and may be interpreted in a linear algebra sense. An immediate consequence of the theorem is, therefore, that this maximum number of linearly independent circuits forms a basis for circuits in a graph. In other words, the vector
corresponding to any circuit in the graph is expressible as a linear combination of the vectors in the basis. Figure 6 is a program flowgraph and a basis set of circuits for that flowgraph appears in Figure 7. The circuit (ABEA (BE) \textsuperscript{+} 3 FA) has the corresponding vector 2004200111, which is obtained by the addition of the vectors corresponding to (ABEFA), 2(BEB), and ABEA. (2(BEB) is the vector 0002200000.)

The application of cyclomatic number to program flowgraphs is based on the observation that adding an arc from the exit node to the entry node in any flowgraph produces a strongly connected directed graph. Since it is often of interest to analyze program modules (and their corresponding flowgraphs), the number of the strongly connected components of the flowgraphs of interest is often one. Thus the following definition applies the cyclomatic number concept to program flowgraphs:

**Definition:** The cyclomatic complexity $C(G)$ of a flowgraph $G$ with $n$ nodes and $e$ edges is

$$C(G) = e - n + 2$$

The restriction to one strongly connected component is not severe, since it is clear that a program $P$ consisting of
Flowgraph with Edges Numbered

Figure 6

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ABEFA)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(BEB)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(ABEA)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(ACFA)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(ADCPA)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Basis and Corresponding Vectors for Flowgraph in Figure 6

Figure 7
modules represented by flowgraphs $G_1$ and $G_2$ will have cyclomatic complexity $C(P) = C(G_1) + C(G_2)$.

McCabe uses Mills' result [MIL72] that if $f$, $d$, and $c$ are the numbers of function, decision, and collecting nodes, respectively, in a flowgraph $G$, then the number of edges is

$$e = 1 + f + 3d$$

and the number of nodes is

$$n = f + d + 2$$

so that the cyclomatic complexity of $G$ is

$$C(G) = d + 1.$$ 

The flowgraph constructs used by Mills do not correspond exactly to those used by McCabe, but the discrepancy does not affect the basic result. This relationship between $C$ and $d$, the number of predicates in a program, obviously simplifies the computation of $C$ for any program.

There is one important weakness in McCabe's presentation which must be addressed. The results in Berge's book concerning cyclomatic number for directed
graphs do not insure that the circuits which form a basis of a directed graph are such that each arc in the circuit is traversed in "a forward direction". In other words, the graph theoretic results do not insure that the circuits in a basis are directed circuits. For arbitrary directed graphs this is actually an obvious result, since many directed graphs contain no directed circuits. Therefore, in order for cyclomatic number to be a reasonable measure for program flowgraphs, it must be the case that "flowgraph" sufficiently restricts "directed graph" to insure that a basis of directed circuits can be obtained for any flowgraph G. This result is established in Chapter 4.

McCabe's work has generated a good deal of research interest. G. J. Myers in [MYE77] demonstrated an anomaly of the behavior of cyclomatic complexity as computed using the number of decisions in a program. Myers contends that any adequate program complexity measure should produce the ordering $8a < 8b < 8c$ for the program segments in Figure 8. If one computes cyclomatic complexity counting entire predicates (simple or compound) as one decision then $C(8a) = 2$, $C(8b) = 2$, and $C(8c) = 3$, and it is certainly not the case that $2 < 2 < 3$. On the other hand, if one counts each simple predicate as one decision then the cyclomatic complexities are $C(8a) = 2$, $C(8b) = 3$, and $C(8c) = 3$. Again, the measures fail to correctly establish the proper
ordering.

a. IF X = 0 THEN ...
   ELSE ...

b. IF X = 0 & Y > 1 THEN ...
   ELSE ...

c. IF X = 0 THEN
   IF Y > 1 THEN ...
   ELSE ...
   ELSE ...

Alternative Constructs of Varying Complexity

Figure 8

Myers suggests using an interval to quantify control flow complexity. The lower bound of the interval is the number of decision statements plus one, which corresponds to decisions consisting of entire predicates. The upper bound of the interval is the value obtained by adding one to the number of simple predicates. Myers notes that this approach solves the problem presented by the program segments in Figure 8 since \((2,2) < (2,3) < (3,3)\) seems clear. However, the suggested interval measure would render a program with interval \((2,3)\) incomparable with one of interval \((1,5)\). This is clearly undesirable for a general measure.
The underlying implication of Myers' observation is that control flow complexity is not the only contributing factor to program complexity. It is not simply the arrangement of nodes and arcs in a flow graph which determines a program's complexity, but also the information which is contained in the nodes. W. J. Hansen followed this line of reasoning in suggesting that cyclomatic number and operator count be used to measure program complexity [HAN78]. Hansen supports his measure with a limited number of examples. However the problem of comparing \((a, b)\) with \((c, d)\) where \(a < c\) and \(d < b\) is even more difficult here than with the interval approach. But Hansen's work can be viewed as an early attempt to synthesize cyclomatic number with software science-like measures, which is exactly the approach taken in Chapter 5. The need for this approach is also supported by J. L. Elsnoi and M. Marcotty in [ELS78B] in which the point is again made that control flow complexity is not the sole contributor to program complexity.

This section concludes the review of the body of research which forms the basis of the subsequent analytical evaluations of the measures \(E, K, NN,\) and \(C\) and the synthesis of a new program complexity measure.
Principles of program modularity provide one basis for analytical evaluation of the software effort measure $E$. In its broadest sense, program modularity is a result of a particular approach to problem solving and is expressed in terms of program subunits that are developed independently and then interfaced. At a more basic level, modularity is achieved by isolating frequently occurring sequences of similar code. It is from this latter notion of modularity that the principles used in this analysis of $E$ are obtained. The principles are embodied in two general types of program transformations:

1. Transformations which eliminate repetitions of identical expressions by introducing a new program variable.

2. Transformations which eliminate repetitions of
similar code sequences by implementing a subprogram for those sequences.

Other researchers have noted that there are rather stringent limitations on the program environments which will permit these types of transformations (e.g. [GOR77]). Therefore, in order that the transformations be well defined, the program environment for each modularizing transformation is precisely characterized. As was stressed in Chapter 1 when the analytical methodology used in these studies was presented, the purpose of defining the transformations is not so they may be applied after a program is written, but rather to carefully delineate the types of programming decisions used in the analysis. Thus, for each program transformation used as a basis of evaluation, the following steps are carefully followed:

1. Precisely describe the program transformation, including the conditions under which it is applicable.
2. Analyze the effect of the transformation on the software effort measure E.
3. Compare and contrast the results from 2. with the programming principles which the transformation represents.

In order to accomplish the first step it is necessary to make use of some facility for describing the execution order of the statements in a program. The notation and
results developed by L. D. Fosdick and L. J. Osterweil (FOS76) in their work on data flow analysis are adequate for this description. These researchers were interested in determining when variables in programs are misused. To this end they used flowgraphs and actions on variables to characterize the behavior of operands during the execution of a program. These variable actions are:

1. reference. When at a particular point in program execution the value of a variable must be obtained from memory, then that operand is said to be referenced at that point in the program.

2. defined. When a value is assigned to a variable at a particular point in the execution of a program then that operand is said to be defined at that point.

3. undefined. When at some point during program execution a variable can no longer be referenced then that operand is said to be undefined at that point.

For instance, in a program written in a block structured language, when the block in which a variable is declared is exited, then that variable is said to be undefined at the point of exit.

Although the notion of "flowgraph" need not be quite so restrictive for applicability in data flow analysis, any subsequent references to the term will be based on the definition which appears in Chapter 2.
From these flowgraphs Fosdick and Osterweil build path expressions (which are regular expressions) to represent the behavior of a single variable along its possible execution paths through the program. For the purposes of this analysis these path expressions can be considered to be composed of r's (reference), d's (define), and u's (undefine). For example, given the sequence of assignment statements

\[ A = B + C; \]
\[ A = A + 1.0; \]

the path expression for A would be drd. With this formalism for path expressions it is possible to specify certain undesirable variable behavior. In particular, the following actions on a variable during program execution are indications of problems with the code:

1. definition - definition (dd). Either the first definition is superfluous or there is something missing in the code.
2. undefined - undefined (uu). This sequence is nonsensical (and was not actually covered by Fosdick and Osterweil). It will be used to exclude some possible path expressions in this analysis.
3. definition - undefined (du)
Fosdick and Osterweil labelled these undesirable sequences of actions on variables "data flow anomalies".

Using the ideas developed in data flow analysis and software science research, three specific types of similar sequences of code which may be reduced by commonly available language features are analyzed. The first type involves identical expressions which may be reduced by temporary variable definition; the second type encompasses code sequences which may be reduced by defining a subprogram whose arguments are simple variables or constants; the final type expands the second type by eliminating the restriction on the arguments to subprograms. In each case an examination is made of the effect on the effort measure of transforming programs with similar subsequences. These quantitative results are compared with commonly accepted programming principles which address the handling of repeated sections of similar code. In the actual analysis the behavior of the software science volume measure \( V \) is considered. This approach permits conclusions about \( E \) since, by definition (see Section 2.2) \( E \) is directly proportional to \( V \). The problems noted earlier in Section 2.2 concerning the variability of components of other estimates of \( E \) also motivates the use of the definition. After the three types of similar subsequences are analyzed,
the results are compared and some suggestions for improving the counting strategy are made.

3.1. Common Subexpressions

The term "expression" denotes a sequence of operands and operators which, by a well defined set of rules, can be evaluated to a single value. No explicit definition of variables occurs in an expression (expressions don't contain statements), but side effects resulting in definitions of a variable may (unfortunately) occur. However, with this characterization of "expression", rather intricate code sequences such as ALGOL-60 designational and boolean expressions involving IF clauses are acceptable, since it is the case that these expressions are not made up of entities which are statements in the language. Using this notion of "expression" the general problem of common subexpressions can be analyzed using the following definition:

**Definition**: Let $S$ be a sequence of code with $n_1$ distinct operators, $n_2$ distinct operands, $N_1$ operator occurrences, and $N_2$ operand occurrences. Any subsequence of $S$ is defined to be an occurrence of a **common subexpression** if:

1. The subsequence is repeated, identically, at
least once, anywhere in S.

2. The subsequence is an expression in the language.

3. For each variable v which appears in the subsequence consider the path expression for v corresponding to the entire code sequence. Let E represent that part of the path expression corresponding to an execution of the repeated subsequence and let R1 stand for an arbitrary path expression, possibly null. No portion of the path expression may be of the form R1 E R2 d R3 E R4 or of the form R1 E R2 u R3 E R4. Furthermore, E must be of the form r+ (= r r*).

Informally, the third criterion insures that the value of each variable operand appearing in the common subexpression does not change between the time one occurrence of the subsequence is executed and the time another is executed. In other words, no variable appearing in the repeated subsequence can be defined or undefined between one execution of the common subexpression and a subsequent execution of the common subexpression. Note that the data flow analysis approach to the description of the execution characteristics of the entire code sequence will account for cases where one occurrence of the subexpression is in a
looping construct—the pattern expression can be written to contain an \( E \) for each execution of the subsequence. Also evaluation of the expression itself must not affect the value in, or the accessibility of, any variable appearing in the expression. The latter condition addresses the situation in which a function invocation, say \( F(A) \), alters the value in \( A \).

If a code sequence contains a common subexpression then modifying the code by first preceding the first occurrence (executionally) of the subexpression by an assignment of the value of the expression to a new operand and then replacing each occurrence of the subexpression by the newly defined operand will produce an executionally equivalent sequence of code. The truth of this claim follows immediately from the criteria contained in the definition of a common subexpression—since no operand appearing in the common subexpression can be altered between the first and final evaluations of an occurrence of the subexpression, the use of a temporary variable as described above cannot affect the program behavior.

Given the above characterization, an examination of the effect of the reduction of common subexpressions proceeds readily. Let
105

\[ j = \text{the number of common subexpression repetitions} \]
\[ m = \text{the length of the common subexpression} \]

and let the symbol "^n" denote software science measures after subexpression reduction. Then

\[ \hat{n}_1 = n! \text{ (assuming other occurrences of the assignment operator appear in the original code sequence)} \]
\[ \hat{n}_2 = n_2 + 1 \text{ (for the new temporary operand)} \]
\[ \hat{N} = N + (m + 3) - (j + 1)(m - 1) \text{ (adding occurrences of the new operand, the assignment operator, the subexpression of length } m \text{, and an end-of-statement operator for the temporary variable definition, and then deducting } j+1 \text{ occurrences of the subexpression replaced by a single operand)} \]
\[ = N + 4 - j(m - 1) \]
\[ \hat{V} = [N + 4 - j(m - 1)] \log(n + 1) \]

From these observations it follows that the volume of the new code sequence will be less than that of the original sequence if and only if the number of common subexpression repetitions, \( j \), is such that

\[ j > \frac{(N + 4 - N \log n / \log(n + 1))}{(m - 1)} \]
Thus the threshold value for the number of repetitions needed in a code sequence in order to achieve volume reduction is a function of the number of types on the original code sequence, the number of tokens in the code sequence, and the length of the common subexpression.

In an effort to determine threshold values for $j$ using measures from actual programs, a program was developed to analyze previously available operator and operand counts for 154 PL/I programs [ELS76] and 425 FORTRAN programs [BUL74B]. The rows of Table I correspond to particular subexpression lengths and the columns to number of repetitions. The table entries are numbers of programs from the data set which, for the subexpression length of that row, require the number of subexpression repetitions of that column. For instance, for subexpressions of length nine, in each of the 579 programs only one repetition of such subexpressions would be required to insure that temporary variable definition would result in volume reduction. In the lower bound case in which the common subexpression is of length three, then four subexpression repetitions would be needed before temporary variable definition would insure volume reduction. These quantitative results agree with accepted approaches to common subexpressions—that it is worthwhile to avoid repeating large expressions or frequently repeated expressions, while repetition of short
subexpressions may not be detrimental [KER74B].

Table 1

Effect of Common Subexpression Reduction on Software Volume

<table>
<thead>
<tr>
<th>Common Subexpression Length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>539</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>539</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>579</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>107</td>
<td>472</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>539</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>578</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>579</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Entries in the matrix represent numbers of programs which, for the given common subexpression length, require the indicated number of subexpression repetitions in order to insure volume reduction.

3.2. Simple Common Subsequences

It is possible to conduct an analysis of reduction of similar code sequences via subprogram definition in an analogous manner. For this case the notion of similar subsequences is expanded in the following definition:
Definition: Each element of a set of disjoint subsequences of an arbitrary code sequence $S$ is an occurrence of a simple common subsequence (scs) if:

1. For each pair of subsequences in the set, the subsequences are identical in terms of their patterns of operators.
2. For each pair of subsequences in the set there exists a mapping $C : V_1 \rightarrow V_2$ where $V_1$ and $V_2$ are the sets of operands appearing in the first subsequence of the pair and the second subsequence of the pair, respectively. Furthermore, if $v_i \in V_1$ and $v_j \in V_2$ and $C : v_i \rightarrow v_j$ and $v_i$ appears in the $r$th token position of the first subsequence, then $v_j$ must appear in the $r$th token position of the second subsequence.
3. The elements of the set of subsequences are in fact sequences of simple statements and complete compound statements—a subsequence boundary must not break up a compound statement.
4. The subsequences are each situated in the entire code sequence in a way such that each subsequence is only executed by control flowing into the first statement in the subsequence and is only exited by control passing implicitly from the last statement in the subsequence.
There are several aspects of this restrictive definition which warrant some explanation. Condition 1 necessitates that GO TO statements and their labels be handled as special cases. It is possible that two subsequences s1 and s2 satisfy the definition of scs's except that s1 contains the statement GO TO <label i> and, in the corresponding token position, s2 contains GO TO <label j>. Now if <label i> labels the "same" statement in s1 as does <label j> in s2, then s1 and s2 should be considered scs occurrences. This can be achieved by including statement labels as elements of the domain and range of the mapping C in condition 2. The GO TO <label i> will be considered as identical to GO TO <label j> for the purpose of satisfying condition 1 if C: <label i> → <label j>. Note that <label i> and <label j> must be contained in their respective subsequences in order for condition 4 to be satisfied.

Condition 1 also imposes a restriction on operators which precludes defining subprograms with parameters whose arguments are subprogram names, since subprogram names are components of invocation operators and the patterns of operators in occurrences of a scs must be identical. This problem could be overcome by adding tokens which are subprogram names to the domain and range of the mapping function C. If condition 2 is still satisfied with this expanded domain and range then a scs is identified. This
extension is omitted from the definition because of the complexity it adds to the characterization of operand parameterization in scs subprogram definition. Program transformations that the programmer might be expected to recognize in the design process before the final code is actually generated are the objects of interest. For this reason, the limitations don't appear to lessen the utility of the definition for this purpose.

The PL/I program which appears in Figure 9 illustrates several of the features of the definition of a scs. The program accepts as input pairs of 30-character-long strings which are stored in parallel arrays. It then sorts the pairs by the second member and outputs the arrays. There are two pairs of subsequences of code in the program which satisfy the definition of a scs. The first occurrence of the first scs begins with statement number 18 and extends through statement number 33. The additional, disjoint occurrence of that subsequence begins with statement 38 and extends through statement 53. The second subsequence is identical to the first in terms of its pattern of operators and the mapping of operands is:

```c
C = (SORTED, SORTED), ('NO', 'NO'),
(ALPHABETIZE_NAMES,
ORDER_BY_COLLEGE), ('YES', 'YES'), (CHECK_PAIRS,
PAIRS_CHECKED), (INDEX,INDEX), (1, 1), (NO_OF_ALUMNAE,
NO_OF_ALUMNAE), (NAME, COLLEGE), (HOLD_NAME,
HOLD_COLLEGE), (HOLD_COLLEGE, HOLD_NAME), (COLLEGE,
```
The position criteria for the mapping of operands is satisfied. The second scs extends from statement number 35 to statement number 37. This subsequence is repeated identically in statements 55 through 57 and hence the definition is trivially satisfied.

### Program Containing Two Simple Common Subsequences

**Figure 9**
Figure 9 (cont).

STMT SOURCE TEXT

24 SORTED = 'NO';
25 HOLD_NAME = NAME (INDEX);
26 HOLD_COLLEGE = COLLEGE (INDEX);
27 NAME (INDEX) = NAME (INDEX + 1);
28 COLLEGE (INDEX) = COLLEGE (INDEX + 1);
29 NAME (INDEX + 1) = HOLD_NAME;
30 COLLEGE (INDEX + 1) = HOLD_COLLEGE;
31 END;
32 END CHECK_PAIRS;
33 END ALPHABETIZE NAMES;
34 PUT PAGE EDIT ('LIST OF ALUMNAE', 'NAME', 'COLLEGE',
   ' ', ' ') (COL (30), A, SKIP (2),
   (2) (COL (18), A, COL (58), A, SKIP (0)));
35 DO INDEX = 1 TO NO_OF_ALUMNAE;
36 PUT SKIP EDIT (NAME (INDEX), COLLEGE (INDEX))
   (A, COL (57), A);
37 END;
38 SORTED = 'NO';
39 ORDER BY COLLEGE:
   DO WHILE (SORTED = 'NO');
   SORTED = 'YES';
   PAIRS_CHECKED:
     DO INDEX = 1 TO NO_OF_ALUMNAE - 1;
     IF COLLEGE (INDEX) = COLLEGE (INDEX + 1) THEN
     DO;
       SORTED = 'NO';
       HOLD_COLLEGE = COLLEGE (INDEX);
       HOLD_NAME = NAME (INDEX);
       COLLEGE (INDEX) = COLLEGE (INDEX + 1);
       NAME (INDEX) = NAME (INDEX + 1);
       COLLEGE (INDEX + 1) = HOLD_COLLEGE;
       NAME (INDEX + 1) = HOLD_NAME;
     END;
     END PAIRS_CHECKED;
   END ORDER BY COLLEGE;
   PUT PAGE EDIT ('ALUMNAE LIST ORDERED BY COLLEGE',
      'NAME', 'COLLEGE', ' ', ' ')
      (COL (25), A, SKIP (2), (2) (COL (18), A,
      COL (58), A, SKIP (0)));
35 DO INDEX = 1 TO NO_OF_ALUMNAE;
36 PUT SKIP EDIT (NAME (INDEX), COLLEGE (INDEX))
   (A, COL (57), A);
37 END;
38 END LISTS;
Given an arbitrary scs in a sequence of code it may be possible to implement that subsequence as a subroutine subprogram and then simply invoke the subprogram at each place in the original sequence where the scs occurred. (Function subprograms are covered later in the chapter.) Whether or not such an implementation is possible and the form of the subprogram definition are influenced by features of the language in which the code is written and by the nature of the original sequence. Language features which are pertinent to these considerations are:

1. Operator definition facilities. This type of facility would make the determination of identical operators quite difficult. Snobol 4's OPSYN feature is a case in point. (Such a feature would, in fact, render the definition of a scs inadequate.)

2. Parameter/argument correspondence. Any language restricted exclusively to call-by-value will not, in general, support scs reduction via subprogram definition since operands may be defined in the scs. Other common techniques for parameter/argument correspondence (call-by-reference, call-by-value-result, call-by-name) do permit the substitution of an equivalent subprogram for an scs. This last claim follows from the fact that the definition of scs's excludes invocation of subprograms involving arguments
other than simple variables or constants and the definition leads to subprograms which are only exited after executing the last statement of the scs.

The aspect of subprogram definition which is influenced by the form of the original code sequence is the determination of which operands must be parameterized. Suppose that a particular code sequence contains a scs and that C1 vi -> vj where vi is an operand contained in the first occurrence (executionally) of the scs and vj is an operand contained in a subsequent execution of an occurrence of the scs. If vi or vj is a variable and the behavior of vi or vj (as characterized using data flow analysis) is such that:

1. the operand is first defined in an execution of an occurrence of the scs and subsequently is first referenced after that execution of the scs, or
2. the operand is first referenced in an execution of an occurrence of the scs, or
3. the operand is first undefined in an execution of the scs (and then, presumably, defined in that execution of the scs) and then is first referenced after that execution of the scs,

or if vi and vj are constant and not equal, then the operand position occupied by vi and vj cannot be implemented as a local variable in the definition of a subprogram for the
scs. In all other cases for the behavior of the variables appearing in the scs occurrences, implementation of vi and vj as a local variable, with any name unique to the subprogram, is possible. The following lemma and theorem establish this result formally.

**Lemma:** Suppose that a code sequence contains a scs and that C; vi + vj where vi is contained in the first occurrence of the similar subsequence and vj occurs in a repetition of that subsequence. Furthermore, let the path expression for vi be R1 E1 R2 where E1 corresponds to the behavior of vi in the scs and let the path expression for vj be R3 E2 R4 where E2 corresponds to the behavior of vj in the scs repetition. Then E1 = E2.

**Proof:** This result follows immediately from the requirement of identical operator patterns and the position criteria for operands.

**Theorem:** Assume there is a code sequence with a scs, mapping of operands, and path expressions as in the previous lemma. Since E1 = E2 the path expression for vi may be expressed as R1 E R2 and that for vj as R3 E R4. If in either the path expression for vi or the path expression for vj we have: 
1. the operand first defined in the execution of the scs and subsequently referenced after the execution of the scs,
2. the operand first referenced in the scs, or
3. the operand undefined in the scs (then, presumably, defined in the scs) and then referenced

or if \( v_i \) and \( v_j \) are unequal constants, then the operands \( v_i \) and \( v_j \) cannot be implemented using a local variable in the subprogram definition. In all other cases the operand may be local and either \( v_i \) or \( v_j \) (or any other unique variable name) may be used in the subprogram definition.

**Proof**: The case for \( v_i \) and \( v_j \) being constant is clear. Now, \( R_1 \) and \( R_3 \) have the following possible forms:

- \( R_d \): the last effect on the operand (\( v_i \) or \( v_j \)) before execution of the scs is that it is defined.
- \( R_r \): the last effect is reference
- \( R_u \): the last effect is undefined
- \( \emptyset \): the operand is not affected prior to execution of the scs.

\( E \) may have the form \( rR, dR, \) or \( uR \) and \( R_2 \) and \( R_4 \) may have one of the forms \( dR, rR, uR, \) or \( \emptyset \). From these
possibilities there are 48 path expression forms for $R_1 E R_2$ and for $R_3 E R_4$. Of these possible forms, many can be eliminated from consideration because they contain data flow anomalies—dd, ur, uu, du, or $\emptyset$. The possibilities which do not contain such patterns are (allowing $R$ to represent an arbitrary path expression):

1. $Rr dR dR$
2. $Rr dR uR$
3. $Rr dR \emptyset$
4. $Rr dR rR$
5. $Ru dR aR$
6. $Ru dR uR$
7. $Ru dR \emptyset$
8. $Ru dR rR$
9. $\emptyset dR dR$
10. $\emptyset dR uR$
11. $\emptyset dR \emptyset$
12. $\emptyset dR rR$
13. $Rd rR dR$
14. $Rd rR uR$
15. $Rd rR \emptyset$
16. $Rd rR rR$
17. $Rr rR aR$
18. $Rr rR uR$ First $R$ must be $R_1 d R_2$.
19. $Rr rR \emptyset$
20. $Rr rR rR$
21. $Rr uR dR$
22. $Rr uR uR$ First $R$ must be $R_1 d R_2$
23. $Rr uR \emptyset$ and second must be $d R_3$.
24. $Rr uR rR$

Now consider the following three cases:
1. $E = dR$. If either the path expression for $v_i$ or the path expression for $v_j$ is of form 4, 8 or 12, then these operands cannot be local to the subprogram definition. In all other cases in which the first action on $v_i$ and $v_j$ in the execution of the scs is define, the operands may be local (and the operand used in the subprogram definition may be either $v_i$ or $v_j$).

2. $E = rR$. In all cases in which the first action on $v_i$ and $v_j$ is reference, the operands cannot be local (forms 13-20).

3. $E = uR$. In this case $R$ must be of the form $dR'$. Forms 21-23 may be implemented by a local variable, while form 24 cannot (due to the subsequent reference). Q.E.D.

There is one special situation in which the program environment dictates that variable operands can be implemented as local variables according to the criteria, but they might also be implemented as global to the defined subprogram—when they are in fact the same operand (i.e., $C$ maps $v_k$ onto itself). How $v_i$ and $v_j$ must be handled in the first three cases which specify when an operand cannot be made local depends on whether $v_i = v_j$. If they are the same variable, then they may be made global to the defined subprogram, as long as the particular language supports such
a facility. If the language does not support global variables of some sort, then $vi$ and $vj$ must be parameterized. If $vi$ is not equal to $vj$ then these operands must be parameterized in the subprogram definition. This discussion is summarized by the pseudo-English algorithm contained in Figure 10.

Let $vi$ and $vj$ be operands from disjoint occurrences of a scs such that $C : vi \rightarrow vj$. Then a determination of the disposition of the operands $vi$ and $vj$ as a subprogram definition may be made by:

```
if $vi$ and $vj$ may not be implemented as a local variable
   then
       if $vi = vj$
           then they may be global
           else they must be parameterized
       else
           if $vi = vj$
               then they may be global
               else they may be parameterized
```

Disposition of Operands in Subprogram Definition of a Scs

Figure 10
LISTS: PROCEDURE OPTIONS (MAIN);
DECLARE (NAME, COLLEGE) (50) CHARACTER (30);
DECLARE (EOF_FLAG,
    INDEX, /*GENERAL PURPOSE ARRAY INDEX*/
    NO_OF_ALUMNAE)
    FIXED BINARY (31, 0);
DECLARE (HOLD_NAME, HOLD_COLLEGE) CHARACTER (30);
SORT:
PROCEDURE (ORDER, PARALLEL);
DECLARE (ORDER, PARALLEL) (*) CHARACTER (*);
DECLARE SORTED CHARACTER (3) VARYING;
DECLARE (HOLD_ORDER,
    HOLD_PARALLEL)
    CHARACTER (30);
SORTED = 'NO';
ALPHABETIZE ORDER:
    DO WHILE (SORTED = 'NO');
        SORTED = 'YES';
        CHECK_PAIRS:
            DO INDEX = 1 TO NO_OF_ALUMNAE - 1;
                IF ORDER (INDEX) ORDER (INDEX + 1) THEN
                    DO;
                        SORTED = 'NO';
                        HOLD_ORDER = ORDER (INDEX);
                        HOLD_PARALLEL = PARALLEL (INDEX);
                        ORDER (INDEX) = ORDER (INDEX + 1);
                        PARALLEL (INDEX) = PARALLEL (INDEX + 1);
                        ORDER (INDEX + 1) = HOLD_ORDER;
                        PARALLEL (INDEX + 1) = HOLD_PARALLEL;
                    END;
            END CHECK_PAIRS;
        END ALPHABETIZE_ORDER;
    END SORT;
EOF FLAG = 0;
ON ENDFILE (SYSIN) ELF FLAG = 1;
GET EDIT (HOLD_NAME, HOLD_COLLEGE) (A (30), X (4),
    A (30));
INDEX = 0;
INPUT_ALUMNAE:
    DO WHILE (EOF_FLAG = 0);
        INDEX = INDEX + 1;
        NAME (INDEX) = HOLD_NAME;
        COLLEGE (INDEX) = HOLD_COLLEGE;

Revision of Program in Figure 9

Figure 11
Several aspects of this claim about operand parameterization are demonstrated by the longer scs of the program in Figure 9. There are three operands which, according to the stated criteria, cannot be implemented as local variables. NO_OF_ALUMNAE, COLLEGE, and NAME are first referenced in the scs, and, since C maps NO_OF_ALUMNAE onto itself, NO_OF_ALUMNAE may be implemented as a global
variable (1). COLLEGE and NAME cannot be made global and so must be parameterized. The remaining operands in the scs may be implemented as local variables. However, C maps INDEX onto itself and so INDEX may be made global to a defined subprogram. C also maps SORTED onto itself but SORTED only appears in the scs and hence letting it be local seems reasonable (2). The remaining variables, HOLD_NAME and HOLD_COLLEGE, cannot be made global and are, therefore, implemented as local variables HOLD_ORDER and HOLD_PARALLEL (3). The program with this longer scs implemented as a subprogram appears in Figure II.

This characterization of the program environments in which a particular subprogram definition of a scs is possible describes adequately, for our purposes, such

(1) In the subsequent software science analysis of scs's it is established that implementation of an operand as a global variable is preferred to implementation as a parameter and implementation of an operand as a global variable is, in many cases, preferable to using a unique local variable.

(2) In terms of the software science measures, the counts are the same whether the operand appears in the subprogram or as a global variable in the containing program.

(3) It is clear that HOLD_NAME and HOLD_COLLEGE might also be parameterized and the choice of implementation as local variables will be analyzed in the subsequent discussion.
program transformations. However, if one were interested in pursuing the identification of scs's and the operands which must be implemented as local variables in subprogram definition of scs's, then it would be worthwhile to consider other program transformations which would produce scs's. If a scs could be produced by changing the order of two "unrelated" assignment statements such as

\[
\begin{align*}
A &= 01 \\
B &= C + D1
\end{align*}
\]

then such a transformation might be worthwhile. No claim is being made that the discussion of the program environments presented here is sufficient to handle the problem of scs identification, but that is not the concern of this particular research. The objective is, again, to investigate the effects on programming difficulty of different approaches to writing programs, and this is accomplished by comparing programs written using one approach and an "opposing" approach. It is quite important to characterize the environments in which the programs under consideration are comparable, but the concern here is not to actually produce one program from the other. The hope is to demonstrate the utility of certain software science measures by showing that their behavior supports certain programming
principles in some well described programming environments.

There are three questions pertinent to the stated goals which arise from the preceding characterization of the environments in which scs's may be implemented as subprograms in a program revision:

1. Suppose we have a scs in which \( C: v_i \rightarrow v_i \) where the operand may be local to a defined subprogram for that scs. Is it "better" to implement \( v_i \) as a local variable in the subprogram or simply allow \( v_i \) to be global?

2. Suppose we have a scs in which \( C: v_i \rightarrow v_i \) and the operand cannot be local to a defined subprogram for the scs. Is it better to parameterize \( v_i \) or allow \( v_i \) to be global?

3. Under what conditions is it better to implement a scs as a subprogram?

A quantification of "better" based on the software science effort measure is investigated and the results are compared to principles concerning these situations.

The first question may be addressed by considering an arbitrary code sequence \( s \) with \( n_1 \) distinct operators, \( n_2 \) distinct operands, \( N_1 \) operator occurrences, and \( N_2 \) operand occurrences. Also assume \( s \) contains \( j \) repetitions of a scs with length \( m \), \( p \) operands which must be parameterized, and
one operand \( vi \) which may be implemented as a local variable, but which may also be global to the defined subprogram (i.e., \( C: \ vi \rightarrow vi \) for every occurrence of the scs and \( vi \) may be local in a defined subprogram according to the criteria summarized above). There are two cases which must be considered:

1. \( vi \) appears in \( s \) outside the scs occurrences. The effects on the software science measures of implementing \( vi \) as local to the subprogram would be:

\[
\begin{align*}
\hat{n} &= n_l + 1 \quad \text{(the invocation is counted as a new operator)} \\
\hat{n}_2 &= n_2 + p + 1 \\
\hat{N} &= N + m - (j + 1)(m - 2p - 2) \quad \text{(the subprogram definition requires \( m \) tokens and then for \( j + 1 \) occurrences for the scs we replace \( m \) tokens with the subprogram invocation operator, the argument grouping operator, \( p \) arguments, \( p - 1 \) argument separators, and an end-of-statement operator)} \\
\hat{V} &= [N + m - (j + 1)(m - 2p - 2)] \log(n + p + 2) \\
\end{align*}
\]

(Eq. 1)

If \( vi \) is implemented as global then no new operand is defined so that the effects on the measures would be:
\( n1 = n1 + 1 \)
\( n2 = n2 + p \)
\( N = N + m - (j + 1) (m - 2p - 2) \)
\( V = (n + m - (j + 1) (m - 2p - 2) \log (n + p + 1) \)

(Eq. 2)

and so allowing vi to be global to a defined subprogram for the scs will require less programming effort since \( V \) from equation 1 is greater than \( V \) from equation 2. This result agrees with commonly held notions about variables in that the use of a local variable would create synonymous operands which have been identified as a program impurity [BUL73 and HAL73A]. It should be emphasized that inherent in the foregoing analysis is the assumption that implementing vi as a global is feasible. There are languages and programming environments (e.g., a programming team working on a large project) where this assumption may not hold.

2. vi appears only in the scs. In this case implementation as a local variable will result in the measures:

\( n1 = n1 + 1 \)
\( n2 = n2 + p \) (a new local variable is used, but the old vi for the entire code sequence is not used)
\[ \hat{N} = N + m - (j + 1)(m - 2p - 2) \]
\[ \hat{V} = (N + m - (j + 1)(m - 2p - 2)) \log (n + p + 1) \]

If \( vi \) is made global then the effects on the measures are the same as if \( vi \) is made local to the defined subprogram, and so the measures indicate no preference (in terms of programming effort) for a global or local implementation of \( vi \).

The answer to the second question concerning the case in which \( C: vi \to vi \) and \( vi \) cannot be implemented as a local variable can be pursued in a similar manner. Suppose \( s \) is a code sequence with measures \( n1, n2, N1, \) and \( N2 \) and \( j \) repetitions of a scs of length \( m \). Also suppose that the scs in \( s \) has \( p - 1 \) operands (other than \( vi \)) which must be parameterized and \( q \) operands which are made local to the subprogram for the scs. Then the effects on the relevant measures of parameterizing \( vi \) are:

\[ \hat{n1} = n1 + 1 \]
\[ \hat{n2} = n2 + q + p \]
\[ \hat{N} = N + m - (j + 1)(m - 2p - 2) \] (again, the subprogram definition requires \( m \) tokens and for \( j + 1 \) occurrences of the scs \( m \) tokens are replaced by the subprogram invocation operator, the argument grouping operator, \( p \) arguments, \( p - 1 \) argument separators, and the
end-of-statement operator)

\[ \hat{V} = (N + m - (j + 1)(m - 2p - 2) \log (n + q + p + 1) \]

(Eq. 3)

In the case in which \( \nu_i \) is simply made global to the defined subprogram we have:

\[
\begin{align*}
\hat{n}_1 &= n_1 + 1 \\
\hat{n}_2 &= n_2 + q + p - 1 \\
\hat{N} &= N + m - (j + 1)(m - 2p) \\
\hat{V} &= (N + m - (j + 1)(m - 2p)) \log (n + q + p) \\
\end{align*}
\]

(Eq. 4)

and the obvious implication of equations 3 and 4 for \( V \) is that the implementation of \( \nu_i \) as global to the defined subprogram is preferred. Again this result must be tempered with any particular restrictions on global variables in a particular situation—the claim being supported is simply that implementation of \( \nu_i \) as a global variable rather than as a parameter will require less programming effort.

A similar approach may be used in addressing the more general problem posed by the third question. Suppose again that the sequence is a code sequence with measures \( n_1, n_2, N_1, \) and \( N_2 \) and that \( S \) contains \( J \) repetitions of a scs the sequence of length \( m \). In addition, suppose \( s \) has \( q \) operands
which must be implemented as local variables in a subprogram
definition for s, and that s contains p operands which must
be parameterized. Then the effects on the software science
measures of rewriting the program implementing s as a
subprogram with p parameters and q local variables is:

\[
\begin{align*}
\hat{n}_1 &= n_1 + 1 \\
\hat{n}_2 &= n_2 + p + q \\
\hat{N} &= N + m - (j + 1) (m - 2p - 2) \\
\hat{V} &= \left( N + m - (j + 1) (m - 2p - 2) \right) \log (n + p + q + 1) \\
\text{(Eq. 5)}
\end{align*}
\]

Using equation 5 it follows that \( \hat{V} < V \) if and only if

\[
\begin{align*}
j > \frac{N + 2p + 2 - N \log (n)}{\log (n + p + q + 1)} \frac{1}{(m - 2p - 2)} \\
\text{(Eq. 6)}
\end{align*}
\]

In summary, whether or not reducing scs's by subprogram
definition will result in actual volume reduction is a
function of the number of types in the program, the number
of tokens in the program, the scs length, the number of scs
operands requiring parameterization, and the number of scs
operands which must be implemented as local variables, but
which will still appear in the remainder of the program.
The program in Figure 9 provides a specific case. The relevant software science measures for this program are:

\[
\begin{align*}
n_1 &= 15 \\
n_2 &= 12 \\
N_1 &= 160 \\
N_2 &= 119 \\
V &= N \log n = 1326.6
\end{align*}
\]

and the longer scs has length 91, \(p = 2\), and \(q = 2\). Given these factors, the number of repetitions needed to insure that volume reduction would be achieved by subprogram definition for the scs is the smallest integer greater than .2315. The measures for the revised program, which appears in Figure 11, are:

\[
\begin{align*}
n_1 &= 16 \\
n_2 &= 16 \\
N_1 &= 118 \\
N_2 &= 82 \\
V &= N \log n = 1000.0 \text{ (this agrees with the value predicted by equation 5)}
\end{align*}
\]

The shorter scs in the example has length 18 and would require no parameters and no local variables. In this case
the number of scs repetitions needed to insure that subprogram definition would insure volume reduction is the smallest integer greater than .3139. Thus effort would also be reduced by implementing the shorter scs as a subprogram.

In order to investigate whether inequality 6 corresponds to commonly held notions about modularity in programming, it is necessary to determine its behavior for values of n and N which are typical of actual programs. A program was developed which would, for a given number of operands p requiring parameterization in a scs of length m, determine how many repetitions of a scs would be required in order for subprogram definition of the subsequence to result in volume reduction. The program was run using the FORTRAN and PL/1 data cited earlier. The results for scs's requiring two or four parameters and no local variables are given in Table 2 and Table 3, respectively. Table 2 indicates that, for scs's requiring two parameters for subprogram definition, a scs of length 8 would necessitate eight repetitions to insure volume reduction. A scs of length 22 will insure volume reduction with but one repetition. With four parameters required for subprogram definition, a scs of length 12 would have to be repeated 13 times in order to insure volume reduction, and the minimum
scs length required in order to guarantee that reducing only two occurrences will insure volume reduction is 36. These results are not unreasonable in view of the fact that each invocation of the subprogram with four arguments would be of length 10. One can gain some insight into these results by assuming that the average number of tokens in a statement is four (clearly a minimal assumption) so that, in the case where four operands must be parameterized, a scs consisting of nine statements could be implemented as a suborogram with assured volume reduction. These results support the notion that highly modularized programs are to the programmer's advantage.
Table 2

Effect of Reducing Simple Common Subsequences with Two Parameters and No Local Variables

<table>
<thead>
<tr>
<th>Subsequence Length</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>119</td>
<td>313</td>
<td>469</td>
<td>549</td>
<td>572</td>
<td>573</td>
<td>578</td>
<td>579</td>
</tr>
<tr>
<td>2:</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>313</td>
<td>549</td>
<td>573</td>
<td>560</td>
<td>460</td>
<td>266</td>
<td>110</td>
<td>30</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3:</td>
<td>0</td>
<td>119</td>
<td>530</td>
<td>265</td>
<td>30</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4:</td>
<td>19</td>
<td>430</td>
<td>30</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5:</td>
<td>294</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6:</td>
<td>236</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7:</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8:</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The minimum scs length (m) which will insure volume reductions is 22.
Table 3

Effect of Reducing Simple Common Subsequences with Four Parameters and no Local Variables

<table>
<thead>
<tr>
<th>Subsequence Length</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>14</td>
<td>50</td>
<td>130</td>
<td>246</td>
<td>379</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>50</td>
<td>246</td>
<td>459</td>
<td>554</td>
<td>572</td>
<td>573</td>
<td>565</td>
<td>529</td>
<td>449</td>
<td>333</td>
<td>200</td>
</tr>
<tr>
<td>3:</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>130</td>
<td>455</td>
<td>518</td>
<td>331</td>
<td>120</td>
<td>25</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4:</td>
<td>0</td>
<td>4</td>
<td>242</td>
<td>424</td>
<td>118</td>
<td>111</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5:</td>
<td>0</td>
<td>126</td>
<td>308</td>
<td>24</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6:</td>
<td>4</td>
<td>329</td>
<td>23</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7:</td>
<td>46</td>
<td>109</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8:</td>
<td>196</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9:</td>
<td>213</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10:</td>
<td>95</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11:</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12:</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13:</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The minimum scs length (m) which will insure volume reduction is 36.
Table 3 (cont.)

<table>
<thead>
<tr>
<th></th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>459</td>
<td>515</td>
<td>554</td>
<td>568</td>
<td>572</td>
<td>573</td>
<td>577</td>
<td>578</td>
<td>579</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>64</td>
<td>25</td>
<td>11</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
3.3 Common Subsequences

In the preceding section several restrictions inherent in the definition of simple common subsequences are justifiable due to the complexity of analysis incurred by the removal of these restrictions and the fact that such complexities obviate the likelihood that the programmer would note such entities while designing a program. However, limiting scs's so that only single operands (i.e., simple variables, aggregate names, or constants) are considered in the domain and range of the mapping \( C \) cannot be so readily supported. The implication of the restriction is that arguments to subprograms defined from scs's can only be simple operands, which is unreasonable. To overcome this restriction we again expand the notion of similar sequences of code in the following definition:

**Definition:** Each element of a set of disjoint subsequences of an arbitrary sequence of code is an occurrence of a common subsequence (cs) if each subsequence can be partitioned so that each partition is either a single operator or an expression in the language (possibly a trivial expression consisting of a single operand) and:

1. If the \( i \)th partition (numbering the partitions
of a subsequence from the lexical beginning) of any of the subsequences is the operator \( o_1 \), then the \( i \)th partition of each of the other subsequences is the same operator \( o_1 \).

2. For each pair of subsequences there exists a one-to-one and onto mapping \( C : P_1 \rightarrow P_2 \) where \( P_1 \) and \( P_2 \) are the sets of partitions that are expressions in their respective subsequences. In addition, if \( p_i \in P_1 \) and \( p_j \in P_2 \) and \( C : p_i \rightarrow p_j \) and \( p_i \) appears as the \( r \)th partition of the first subsequence of the pair, then \( p_j \) must appear as the \( r \)th partition of the other subsequence.

3. For each operand \( v_i \) in a partition \( p_i \) that is either of length greater than one or is such that \( C : p_i \rightarrow p_j \) and \( p_j \) is of length greater than one, then \( v_i \) may only be referenced (as opposed to defined and/or undefined) during the execution of the subsequence \( p_i \).

4. The subsequences are in fact sequences of simple or complete compound statements and each subsequence is executed only by control flowing into the first statement of the subsequence. In addition, any explicit transfer of control appearing in any of the subsequences results in branching to statements within the subsequence.
This definition, as was the case with the one for scs's, is constrained in its applicability in particular situations by language features. Similarly, the nature of the subprogram defined from the cs is affected by the specific program environment. A programming language with operator redefinition facilities would render the definition of cs inadequate since condition I relies on the semantics of operator tokens. A language with a calling convention restricted exclusively to call-by-value would make subprogram definition of a cs impossible in cases where operands which are parameterized are defined in their subsequences, although the definition could be restricted to account for this limitation. The program environment affects operand parameterization with cs's in much the same way as it did with scs's. The only modification which must be made for cs's is that if either pi is a partition of an occurrence of a cs and pi has length greater than one or C: pi \rightarrow pj where pj is a partition of another occurrence of the cs and pj has length greater than one, then pi must be parameterized in the subprogram definition—regardless of the results of any data flow analysis of these operands. For all operands appearing in an occurrence of a cs not covered by the preceding case, the parameterization discussion presented for scs's applies.
The definition and subsequent discussion of cs's carefully characterizes the environments in which such similar code sequences may be implemented as a subprogram with invocations in the transformed program in the position originally occupied by each cs occurrence. We noted in the discussion that particular calling conventions must be available in the source language of the program and the possibilities include call-by-dummy-argument, call-by-name and call-by-value-result. In fact, the behavior of a program resulting from the implementation of a cs as a subprogram is not influenced by the source language's use of a particular one of these calling conventions. This result follows from the part of the definition which requires that operands may only be referenced (in the data flow analysis sense) during execution of the cs if they appear in partition positions which have, in at least one occurrence of the cs, a partition of length greater than one. To see that this justifies the claim note that with this condition we could, at the beginning of an execution of an occurrence of the cs, assign each nontrivial expression appearing as a partition in the cs to a temporary variable, and then substitute this variable for each occurrence of the expression, without affecting execution of the cs. Thus, the cs with the nontrivial partitions reduced to simple variables would satisfy the definition of a scs, and it has
already been shown that the calling conventions won't affect program behavior with reduction of scs's. The observation that call-by-dummy-argument behaves like call-by-reference when we are assured that no variable definition occurs for the parameters of interest establishes the correspondence of the calling conventions considered here for cs's to those considered in the previous section for scs's.

For the purpose of analyzing the effect on the software science measures of reducing cs’s via subprogram definition again consider a code sequence with measures n1, n2, N1, and N2. Assume the sequence contains j repetitions of a cs in which p parameters and q local variables are required for the subprogram definition. Let m be the length of the subprogram. In the case of scs’s m was the length of both the scs occurrences and the subprogram, but in the case of cs’s the length of the defined subprogram will be less than or equal to the length of the cs occurrences. Finally, suppose that for the first occurrence of the cs the length of the parameterized partitions is e(1,1) + 1, e(1,2) + 1, ..., e(1,p) + 1. Then

\[ e_l = \sum_{k=1}^{p} e(1,k) \]

is the difference between the length of the subprogram definition and the length of the first cs occurrence.
Define $e_2, \ldots, e_{(j+1)}$ in a similar manner. The effect on the software science measures of defining a subprogram for such a cs would then be:

\[ n_1 = n_1 + 1 \]
\[ n_2 = n_2 + p + q \]
\[ \hat{N} = N + m - (j + 1)(m + ek - (3 + p - l + ek + p)) \] (The subprogram definition requires $m$ tokens and then the $k$th cs of length $m + ek$ is replaced by the invocation operator, the argument grouping operator, the end-of-statement operator, $p - 1$ argument separators, and the argument list with $ek + p$ tokens).

\[ \hat{V} = [N + m - (j + 1)(m - 2p - 2)] \log (n + p + q + 1) \] (Eq. 7)

Equation 7 is identical to equation 5 which characterized $\hat{V}$ for scs's. In other words, suppose there is a scs in a code sequence for which subprogram definition results in volume reduction. Then a similar cs in which any parameterized operand in the scs is replaced by an arbitrary expression may also be implemented as a subprogram with an identical reduction in volume. One might anticipate this result since each expression being parameterized will still appear in the code sequence after subprogram definition as an argument in
some invocation.

3.4 The Counting Strategy

The foregoing analysis of the effects on various software science measures of transforming similar code sequences is based on an operator/operand counting strategy similar to those described in [ELS76]. The extent of agreement between the quantitative results and several commonly advocated programming principles is at least encouraging. However, these studies do reveal a problem with the counting strategy. Suppose the definition of scs's was modified to facilitate simple function subprogram definition—that is, suppose that function subprograms may be defined for expressions and that function subprogram invocations can appear in expressions. (Obviously, the implication is not that function subprograms must be limited to single expressions but this limited case suffices for the present purpose.) Using the notation from Section 3.3, the effects on the software science measures of implementing an expression of length m with no operands which require parameterization (and, clearly, no local variables) would be:

\[ \hat{n} = n + 1 \]
\[ \hat{n}^2 = n^2 \]
\[ \hat{N} = N + m - (j + 1) (m - 1) \]

Under such restricted circumstances it might well be the case that temporary variable definition would suffice. (See Section 3.1 for an explanation of when such an expression is, in fact, a common subexpression.) However, in the analysis for common subexpressions it was established that

\[ \hat{N} = N + m + 3 - (j + 1) (m - 1) \]

which would indicate that \( V \), and hence programming effort, is reduced more by function subprogram definition. This results from counting none of the tokens used to define the function subprogram other than the expression itself. However, programming languages require some tokens to define function subprograms and, while it may be appropriate to ignore such tokens when analyzing the subprogram as a single entity, it seems appropriate to count the entities (e.g., PROCEDURE...END) as grouping operators. The procedure name is similar in many respects to a statement label and thus should not be counted. The results of applying this counting strategy to the 579 programs analyzed previously are summarized in Tables 4 and 5. Clearly, this revision has the effect of requiring slightly more repetitions of a
cs of a given length before subprogram definition will insure volume reduction.

One might carry this counting strategy revision one more step and consider the parameters in the subprogram definitions to be operands which are to be counted. Again, the effect would be to require slightly more repetitions of a given cs before volume reduction will be insured. Both of the counting strategy revisions counter the phenomenon of temporary variable definition requiring more effort than function subprogram definition. The latter suggestion also balances the fact that arguments in invocations are counted and might well be the most reasonable way of counting larger programs.
Table 4

Effect of Revised Counting Strategy for Common Subsequences Requiring Two Parameters

<table>
<thead>
<tr>
<th>Subexpression Length</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>119</td>
<td>313</td>
<td>469</td>
<td>549</td>
<td>572</td>
</tr>
<tr>
<td>2:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The minimum scs length \( (m) \) which will insure volume reduction is 25.
### Table 5

**9 Effects of Revised Counting Strategy for Common Subsequences Requiring Four Parameters**

<table>
<thead>
<tr>
<th>SUBEXPRESSION LENGTH</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>2:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>130</td>
<td>379</td>
<td>515</td>
<td>568</td>
<td>573</td>
<td>578</td>
<td>575</td>
<td>565</td>
</tr>
<tr>
<td>3:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>130</td>
<td>459</td>
<td>554</td>
<td>447</td>
<td>200</td>
<td>64</td>
<td>11</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4:</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>375</td>
<td>438</td>
<td>119</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5:</td>
<td>0</td>
<td>4</td>
<td>365</td>
<td>193</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6:</td>
<td>0</td>
<td>126</td>
<td>189</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7:</td>
<td>0</td>
<td>329</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8:</td>
<td>14</td>
<td>109</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9:</td>
<td>116</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10:</td>
<td>249</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11:</td>
<td>136</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12:</td>
<td>53</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13:</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14:</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15:</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The minimum scs length \((m)\) which will insure volume reduction is 39.
Table 5 (cont.)

<table>
<thead>
<tr>
<th></th>
<th>26</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>130</td>
<td>246</td>
<td>379</td>
<td>459</td>
<td>515</td>
<td>554</td>
<td>568</td>
<td>572</td>
<td>573</td>
<td>577</td>
<td>578</td>
<td>579</td>
</tr>
<tr>
<td>2</td>
<td>449</td>
<td>333</td>
<td>200</td>
<td>120</td>
<td>64</td>
<td>25</td>
<td>11</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
3.5. Summary of Modularity Analysis

The analysis presented in Chapter 3 has been directed toward the determination of whether or not the software science effort measure is reasonably (logically) sensitive to program modularization, at least in so far as modularization is achieved by reduction of similar parts of programs using commonly available language facilities. In both temporary variable definition and subprogram definition the measure agrees with commonly held notions regarding the use of these language facilities in programming. It ought to be noted that there are certainly times when the execution time characteristics of programs written by following those tenets of modularity may be adversely affected, but it has become clear that program performance is affected not only by execution characteristics but also by the programmer-program relationship. The overall result of the analysis is that $E$ is a reasonable program complexity measure when it is evaluated in light of principles of program modularity.
EVALUATION OF MEASURES OF CONTROL FLOW COMPLEXITY

After the successful evaluation of $E$ using modularity principles, it seems appropriate to evaluate $E$ using the same analytical methodology with transformations related to control flow complexity, since there is considerable support for the contention that control flow complexity is also an important software engineering issue. The empirical studies by Sime and Love reviewed in Chapter 2 deal specifically with the extent to which control flow complexity affects programming difficulty. In addition, the three control flow complexity measures presented in Section 2.3 indicate that a number of researchers consider this aspect of program complexity to be important. In fact, a central theme in discussions of structured programming is how to minimize program control flow complexity. Since control flow complexity is an important software engineering issue, any proposed program complexity measure must withstand analysis from the standpoint of those commonly accepted principles of
program control flow.

The methodology which is used to perform this analysis is analogous to that used in the evaluation of $E$ in Chapter 3. Specifically, three transformations which make explicit certain aspects of control flow complexity are utilized. Each of the transformations is carefully defined. Then the difference between $E$ for one program $P$ and $E$ for another program $T(P)$ is considered in light of current accepted notions concerning the desirability of $T(P)$ as opposed to that of $P$. This analysis of $E$ is summarized in Section 4.1. The results of this analysis motivate consideration of other measures of program control flow complexity. The three measures introduced in Section 2.3 are analyzed using the same three transformations in the subsequent sections of this chapter.

4.1. Software Effort

4.1.1. The Linearization Issue

The first transformation used in these analyses involves the translation of a two-dimensional flowgraph into a one-dimensional set of program instructions—the "linearization" of the flowgraph. Clearly, in any linearization it is imperative that the program and
flowgraph compute the same function (i.e. that they be "functionally equivalent"). But much stricter limits can be placed on the linearization process. For example, one can require that the flowgraph and program be "executionally equivalent", which implies that the same input would result in the same sequence of executable steps in both algorithm expressions. An even more restrictive sense of equivalence results from disallowing duplication in the program of the predicates and functions which appear at the nodes of the flowgraph. This sense of equivalence is called "strong equivalence" [LED75].

Even under the restrictions of strong equivalence, there can be many possible linearizations of a particular flowgraph. For example, the program segments which appear as Figures 13 and 14 are both linearizations of the flowgraph in Figure 12, and both program segments are strongly equivalent to this flowgraph. The programs differ in that the code in Figure 13 contains three unrestricted transfer of control operator uses while the code in Figure 14 contains four. It is clear that under the prohibition of predicate and function duplication, differences in linearizations of a particular flowgraph must lie in control structure differences.
IF (A) THEN
  IF (B) THEN D
  ELSE
    e: IF (E) THEN NULL
    ELSE GO TO h
    FI;
  f: F
  ELSE
    IF (C) THEN GO TO e
    ELSE
      IF (I) THEN GO TO f
      ELSE G
      FI
    FI;
  h: H

Linearization of Flowgraph in Figure 12
with Three GOTO's

Figure 13
IF (A) THEN
  IF (B) THEN D; GO TO f
  ELSE GO TO e
FI
ELSE
  IF (C) THEN
    e:
    IF (E) THEN NULL
    ELSE GO TO h
    FI
  ELSE
    IF (I) THEN NULL
    ELSE G; GO TO h
    FI
  FI;
fi:
  F
fi;
h:
  H

Linearization of Flowgraph in Figure 12

with Four GOTO's

Figure 14

The analysis of E using linearization issues differs from previous analyses in that the basis of comparison is two different linearizations and the transformation is from a flowgraph to source code. In other words the comparison is not actually of P and T(P), but rather of T(P)* and T(P)**. Suppose that in the linearization of a flowgraph G, for each construct in G of the form depicted in Figure 15, where G1 is an arbitrary flowgraph, the statement:

DO WHILE(P);
  G1;
is used to linearize that form. Also suppose that predicate nodes in G which do not appear in the form of Figure 15 are implemented using the statement (1):

IF (P) THEN...
ELSE...

Given this reasonable approach to linearizing G, it follows from the restriction of strong equivalence that differences in linearizations can only come from the way in which the arcs in the flowgraph are expressed in the program. In particular, the differences must lie in which arcs are handled by implicit control flow (resulting from particular choices of the statements which follow alternative constructs) and which are handled by explicit GOTO transfers of control.

(1) Generally, predicate nodes are those nodes in the flowgraph with out-degree greater than one. For the purpose of this study these nodes in a flowgraph are also restricted to having out-degree no greater than two. This is done without loss of generality since a node with out degree greater than two can be represented by a sequence of nodes with out degree equal to two.
Since differences in linearizations of a given flowgraph must result from different uses of GOTO statements, the program segments in Figures 13 and 14 may be considered sufficiently general. Since the program segment in 14 contains one more operator type and one more operator occurrence than the segment in 13, the software volume measure $V$ for 14 must be greater. Therefore $E$ is slightly larger for 14 than for 13, reflecting the fact that the program in 14 makes poorer use of implicit control flow. That this is undesirable is supported by the work by B. Baker on linearizing flowgraphs [BAK77]. In her work Baker presents an algorithm which optimizes use of implicit control flow, given certain constraints which insure the
resulting programs are "structured". (The notion of structure used by Baker does not limit programs to D-structures.) This notion of optimal control flow has the effect of minimizing both the number of types and number of occurrences of unrestricted transfer of control statements in programs satisfying the constraints mentioned previously. Baker contends that this optimal use of implicit control flow is desirable, so that the behavior of $E$ for different linearizations must correctly reflect the differences, since for strongly equivalent linearizations of a particular flowgraph the linearization with more GOTO's will have a greater $E$ measure. Thus, once again, $E$ seems to meet analytical criteria for a program complexity measure.

4.1.2. The Node Splitting Issue

The next two transformations used in the analytic evaluation of $E$ come directly from considerations of structured programming. Both transformations may be used in certain instances to produce structured programs from unstructured programs. The first of these is mentioned briefly in Section 2.3 in the description of normal forms of flowgraphs. These flowgraphs are produced by replicating nodes in the original flowgraph—a process referred to as "node splitting". In many instances such transformations may be used to produce flowgraphs in which each subgraph
consists of a single entry arc and a single exit arc—i.e., structured flowgraphs. (Each D-structure has a corresponding flowgraph form and these forms are characterized by having a single entry and a single exit arc.) In fact, Knuth and R. W. Floyd have proved that node splitting alone can be used to structure any flowgraph which does not contain multiple exit loops [KNU74]. W. Wulf has observed that flowgraphs without loops may always be structured via node splitting, since their (at worst) lattice-like structure insures that there are only a finite number of paths from the entry node to the exit node. Thus, each node can appear on but a finite number of paths, so that such flowgraphs may be restructured by simply "enumerating" each path from the entry node to the exit node in a transformed, executionally equivalent version of that original flowgraph [WUL72]. Flowgraphs with lattice-like structure can therefore be expressed as trees by node splitting. Figures 16 and 17 provide a simple illustration of this rather brute force approach to structuring. Because flowgraphs without loops can be so readily structured by node splitting, this type of flowgraph is considered first in the analysis.

The drawback to this brute force approach is illustrated by the code in Figure 18, which is the natural way to express the flowgraph in Figure 16 in a language with
Flowgraph
Figure 16

Tree Corresponding to Flowgraph in Figure 16
Figure 17

IF (A) THEN B
ELSE C

Natural Coding of Flowgraph in Figure 16
Figure 18
Flowgraph
Figure 16

Tree Corresponding to Flowgraph in Figure 16
Figure 17

IF (A) THEN B
ELSE C

Natural Coding of Flowgraph in Figure 16
Figure 18
an IF...THEN...ELSE construct. What is required for an adequate characterization of node splitting on flowgraphs without loops is a structuring procedure which preserves any natural IF...THEN...ELSE structure of the original flowgraph.

Such a procedure must make use of a more formal characterization of "natural IF...THEN...ELSE structure". Again Baker's work on structuring arbitrary flowgraphs provides some assistance (although her structuring algorithms produce flowgraphs which are strongly equivalent to their original forms, so that node splitting is not a part of the procedure Baker presents). The following definitions provide the appropriate formalization:

**Definition**: Node p dominates node q in a flowgraph G, if every path from the start node to q in G must pass through p. If there is no other dominator between p and q, then p is called the immediate dominator of q and this relationship is denoted p = DOM(q).

**Definition**: Suppose p is an IF node in flowgraph G. If DOM(q) = p and q has in-degree greater than or equal to two then q follows p, which is denoted q = FOLLOW(p).
The second definition formally specifies those nodes whose corresponding statements must follow particular IF statements. Baker provides an algorithm for finding follow nodes for all nodes in a flowgraph. Thus the desired node splitting algorithm can make use of this follow node information. The algorithm should only split those nodes with in-degree greater than one which are not follow nodes. Such nodes are referred to as "node splitting candidates".

"Node splitting" is not a precisely defined concept as there are many reasons why nodes in graphs might be split (e.g. see [HEC77]). In each case, however, nodes are split in order to reduce their in-degree. The differences in the actual procedures lie in what is done with the arcs emanating from the split node. For example, consider the flowgraph in Figure 19. By definition, $\text{FOLLOW}(A) = F$ and the IF node $B$ has no follow node. This leaves two node splitting candidates, $E$ and $D$, and these nodes must be split in order to structure the flowgraph. Suppose that $E$ is split first. The flowgraph in Figure 20 is produced if $E$ is replicated for each of its in-arcs and each out-arc from $E$ is also replicated. However, if all the paths from $E$ to $F$ are replicated for each copy of $E$ (and each in-arc to nodes on such paths are replicated), the flowgraph which appears in Figure 21 is produced. Notice that in Figure 21 that two copies of the original node $D$ are node splitting
candidates. Thus when these nodes are split the resulting flowgraph will have four copies of D. However when D is split in Figure 20 only three copies of D will appear in the resulting flowgraph.

Flowgraph with Two Nodes which are Splitting Candidates

Figure 19

One Approach to Splitting Node E in Flowgraph in Figure 19

Figure 20
A Second Approach to Splitting Node E in Flowgraph in Figure 19

Figure 21

Another approach to structuring the flowgraph in Figure 19 is to first split node D and then node E. Again either single arcs or entire paths emanating from a node splitting candidate may be replicated. In this case, either approach yields the flowgraph which appears in Figure 22. Note that this same flowgraph is produced by splitting node D in Figure 20. Thus this example suggests that by only replicating out-arcs of split nodes the node splitting order is unimportant. The example does establish that the order is significant for cases in which nodes are split and entire paths to the exit are replicated. Thus the discussion of this one example suggests the simple procedure which appears as Figure 23.
Procedure STRUCTURE(G,GS)
1. Let GS denote the flowgraph G.
2. Let N be a node splitting candidate in GS.
3. Repeat 3.1 - 3.3 while N ≠ ∅.
   3.1. Produce a new flowgraph by:
      3.1.1. For each arc terminating at N, produce a copy of
              N and give that copy a unique label Ni (i=1, 2, ...

      3.1.2. For each Ni generated in step 3.1.1, produce out
              arcs corresponding to those arcs emanating from
              the original node N in G.
   3.2. Let GS now denote the flowgraph produced in step
       3.1.
   3.3. Let N be a node splitting candidate in GS.
4. Return

Node Splitting Procedure

Figure 23
Notice that STRUCTURE is nondeterministic in its selection of the next node splitting candidate to be split, which corresponds to the observed lack of order noted above. Clearly, if STRUCTURE halts then it returns a structured flowgraph in GS since the procedure iterates until there are no node splitting candidates remaining in GS. That the nondeterminism of STRUCTURE in selecting the next node to be split does not affect the flowgraph which is returned by the procedure is established in the following theorem:

**Theorem:** Assume that STRUCTURE halts when given a flowgraph G without loops as input. Let GS denote an arbitrary flowgraph produced during the execution of STRUCTURE. Then for any choice of node splitting candidate in GS, the same final structured flowgraph is produced.

**Proof:** Let A and B be any two node splitting candidates in GS. Then the theorem is established if it does not matter that STRUCTURE splits A and then immediately splits B, or B and then immediately A. This follows since A and B are arbitrarily chosen node splitting candidates in GS and GS is any flowgraph produced by STRUCTURE when executing on G. There are
two cases to consider:

1. There is an arc from A to B or an arc from B to A (but not both, since there are no loops in G). Suppose the arc is from A to B. Then the situation may be represented as in Figure 24, where there are j arcs into A, k other arcs out of A, n other arcs into B, and m arcs emanating from B. The sequence of flowgraphs produced by splitting A and then B is depicted in Figures 25 and 26. Figure 27 shows the result of first splitting node B and Figure 28 depicts the result of splitting node A in the flowgraph in Figure 27. By splitting node B in the flowgraph in Figure 28 the flowgraph in Figure 29 is obtained, and this is identical to the flowgraph of Figure 26, except for the labels of the replications of the original node B. This establishes that splitting A and then B, or B and then A and then B1 both yield the same flowgraph.

In fact, it is not necessary that B1 be split immediately after A is split. To see this consider the flowgraph in Figure 28 and assume that B1 is not the next choice of STRUCTURE. Eventually B1 will be chosen (node splitting in STRUCTURE decreases the in degree of only the
split node). The only way in which the split of Bl will be different than the way in which it is depicted in Figure 29 is if one or more of the A1 nodes are again split. Suppose that p additional copies of the original A node are produced. Then when Bl is split there will be \( j+p \) in arcs to Bl, yielding \( j+p+n \) copies of B. Now consider the effect of the subsequent splits of the A1 nodes in Figure 29, which depicts splitting Bl immediately after splitting A. Here again p new in arcs to the nodes B(1,1), ..., B(1,J) will be generated, necessitating that p copies of B be produced by splitting those B(1,x) (x=1, ..., J) nodes with in degree greater than one. This will yield the same flowgraph with \( j+p+n \) copies of B. Thus the point at which Bl is split is immaterial as far as the final structured flowgraph is concerned, which establishes case I.
Portion of Flowgraph with Two Node Splitting Candidates

Figure 24

Result of Splitting Node A in Flowgraph in Figure 24

Figure 25
Result of Splitting Node B in Flowgraph in Figure 25

Figure 26

Result of Splitting Node B in Flowgraph in Figure 24

Figure 27
Result of Splitting Node A in Flowgraph in Figure 27

Figure 28

Result of Splitting Node B1 in Flowgraph in Figure 28

Figure 29
2. There is neither an arc AB nor an arc BA. The node splitting procedure STRUCTURE insures that only initial nodes of in arcs to split nodes and terminal nodes of out arcs from split nodes can be influenced in any direct way by a single node split. The condition of case 2 insures that the nodes A and B must be separated by a path consisting of at least one other node. Then clearly it does not matter whether A is split and then B, or vice versa. Q.E.D.

With the proof of the preceding theorem all that remains in order to establish that STRUCTURE provides a sound characterization of node splitting is to show that STRUCTURE halts.

**Lemma**: For any flowgraph G without loops there exits an ordering of node splitting candidates in G which insures that STRUCTURE halts.

**Proof**: An arbitrary ordering is insufficient because splitting a node can create a node splitting candidate which was not a candidate in the original flowgraph (e.g. see Figure 28). This will occur for each node which is at the terminal end of an out arc from a node
which is split. In order for STRUCTURE to halt, it must be the case that this generation of node splitting candidates halts. This is established if there is an order of node splitting candidates which insures that once a node is split, it (or its replications) need never be split again. The following procedure specifies such an order (let $G_i$ denote the flowgraph produced by the $i$th node split):

1. Set $i$ to 0.
2. Repeat steps 2.1--2.3 until there are no node splitting candidates in $G_i$.
   2.1. Perform a breadth first search of $G_i$ to obtain a partial ordering of the node splitting candidates in $G_i$ ([HEC77]). Such an ordering is possible due to the absence of loops.
   2.2. Split (using the approach described in STRUCTURE) node splitting candidate $X$ in $G_i$, where $X$ is less than or equal to any other node splitting candidate according to the partial ordering established in 2.1. (several nodes can be "equal" in their position in a particular order - precisely those nodes which have no strict order relation to one another).
   2.3. Increment $i$. 
This approach to node splitting insures that once a candidate is split its replications cannot subsequently become splitting candidates. Thus the procedure just presented must halt and it yields the desired ordering of the splitting candidates in G. Q.E.D.

Thus the algorithm STRUCTURE performs the desired structuring of an arbitrary flowgraph with a lattice-like structure. With this precise characterization of "node splitting", the analysis of the software science effort measure proceeds readily. STRUCTURE splits nodes in order to avoid explicit transfer of control to the split nodes. This is depicted in Figures 30-33 where the effect of STRUCTURE is to avoid in the program in Figure 33 the "GO TO h" which appears in Figure 32. Now suppose that the node H contains k tokens and that the one explicit transfer of control operator occurrence saved by the node split is, in fact, the only occurrence of that operator. Then, since the software science type count is reduced by one, the effort measure is increased by the transformation if

\[(N + k) \log (n - 1) > N \log n\]

For an appropriately large value of k this will be true, regardless of the values of n and N. However, it would
appear that very unlikely situations must be created in order to find $n$ and $N$ values which would make $k$ nontrivial. In fact, the extensive set of $n$ and $N$ values from the data used in Chapter 3 was used to study the inequality and the largest value of $k$ for which the inequality did not hold was 3. Since any split node will represent more than three tokens in the corresponding program (even a simple assignment statement must have at least 4 tokens), the implication of the analysis is that from the point of view of software effort it is never advisable to node split.

\[
\begin{array}{c}
A \\
B \quad C \\
D \quad H \quad F \\
G \end{array}
\]

STRUCTURE would Split Node H

Figure 30

\[
\begin{array}{c}
A \\
B \quad C \\
D \quad H_1 \quad H_2 \quad F \\
G \end{array}
\]

Result of STRUCTURE on Flowgraph in Figure 30

Figure 31
If (A) Then
  If (B) Then D;
  ELSE Go To h;
ELSE
  If (C) Then
    h: H;
    ELSE F;
  G;

Linearization of Flowgraph in Figure 30

Figure 32

Linearization of Flowgraph in Figure 31

Figure 33

It is worth noting that this result for the well-defined node splitting transformation exemplified by STRUCTURE on flowgraphs without loops is applicable in the more general case of flowgraphs with loops since one would still be trading off the token counts of an entire node for an explicit transfer of control operator. The preceding analysis indicates that this trade off will, for all practical purposes, always result in increased effort.

This behavior of E under node splitting transformations is contrary to what one would expect in an adequate program complexity measure. While no one has argued that node splitting is always advisable, there is considerable support for producing structured programs using node splitting when
It is appropriate. E. W. Dijkstra advocates the desirability of such programs by his first admonition against unrestricted transfer of control [DIJ68]. More recent support for producing (strictly) structured programs is provided by H. Ledgard and M. Marcotty in their survey of formal issues in program control structures [LED75].

4.1.3. Structuring Multiple Exit Loops

The third transformation used to evaluate E also comes directly from issues involving structured programs. As mentioned previously, the one flowgraph construct which cannot be structured using only node splitting transformations is that of a loop with multiple exits. There are two general approaches to structuring such flowgraphs. Unlike the notion of node splitting, both approaches are formally defined in the open literature. E. Ashcroft and Z. Manna have shown that any approach to this problem must involve the use of extra program variables [ASH71]. One approach, also described by Ashcroft and Manna in [ASH71], uses state variables to "remember" key program variable values at "critical points" in the execution of the loop body. These values are then used to determine subsequent paths through a single-entry-single-exit loop which "simulates" the behavior of the original loop. A second approach to structuring multiple exit loops involves
the use of boolean variables. Although a general algorithm for structuring multiple exit loops using boolean variables exists [WUL72], the restrictive notion of "multiple exit loop" depicted in Figure 34 is adequate for the analysis of E, since it represents a simple instance of "multiple exit loop" which is quite readily structured. Other more involved cases have more profound effects on type and token counts. This case is representative of actual situations involving multiple error exits from a loop. The structured flowgraph in Figure 35 is functionally equivalent to the flowgraph in Figure 34. This structured version uses the boolean variable b to predicate the iteration.

Simple Loop with k Exits

Figure 34
As was the case with the node splitting transformation, the behavior of $E$ under the transformation to structure multiple exit loops using boolean variables is inappropriate. Clearly, since this transformation requires an increase in $N$ and no increase in $n$, $E$ will always be increased in a program produced by this transformation. There is considerably more debate on the superiority of programs produced by this approach (i.e. whether the transformed program minimizes programming difficulty) than there is for programs produced by node splitting. Ashcroft and Manna remained neutral on the worth of their structuring algorithm. Ledgard and Marcotty favor this strictly
structured approach [LED75], while Knuth has argued that there are cases better handled by unrestricted transfer of control [KNU74]. Regardless of the final resolution of this discussion, it is clear that there is support for strictly structured programs and the behavior of E contradicts these commonly held opinions. In those cases where the structuring transformation produces a program which minimizes programming difficulty, E must fail to reflect this improvement.

Of the three transformations related to program control flow considered in this analysis, E behaves aberrantly in two of them. This is indicative of possible weaknesses in E with respect to this important factor in program complexity. Consideration of the development of E lends some insight into this problem. The number of distinct operators, nl, does not distinguish between operators. Therefore an occurrence of the operator "*+*" has the same effect on the software effort measure as does an occurrence of the operator "GO TO <label>". It has been demonstrated that numerous researchers consider program control flow complexity to be the primary factor in program complexity and so this even trade-off of "*+*" for GOTO does not seem well motivated.
4.2. Program Knots

The demonstrated weakness of software effort in capturing control flow complexity motivates similar analyses of those program measures which purport to capture control flow complexity. Three such measures are defined in Section 2.3. The first of these measures, program knots \( (K) \), was motivated by a need for a control flow complexity measure which accounts for the linearization issue \cite{W0079}. The program schemas in Figures 36 and 37 are executionally equivalent, given the obvious correspondence of nodes having the same labels. Both schemas are representations of actual programs where the arcs represent transfer of control. Of course the vertical arcs representing natural sequencing would be handled by implicit program control flow. The schema in Figure 36 would be implemented using more explicit transfer of control operators than would the schema in Figure 37, and this difference is properly reflected in the knots measure.
That this appropriate behavior of $K$ is not general is seen in the two linearizations of the flowgraph in Figure 12 which are repeated in Figures 38 and 39 with the knots indicated for both cases. As indicated in the prior discussion of these two linearizations, there is support for
the contention that the program in Figure 38 involves less programming difficulty than the one in Figure 39, because of less complex control flow. Therefore, K does not always appropriately capture differences in linearization.

```
IF (A) THEN
  IF (B) THEN D
ELSE
  IF (E) THEN NULL
  ELSE GO TO h
FI;
FI
FI IF
ELSE
  IF (C) THEN GO TO e
  ELSE
    IF (I) THEN GO TO f
    ELSE G
    FI
    FI
FI
FI
```

Linearization of Flowgraph in Figure 12 which Contains Two Knots

Figure 38
A Different Linearization of Flowgraph in Figure 12
which also Contains Two Knots

Figure 39

The original example of program knots used in Section 2.3 suggests other deficiencies in the measure. The example is repeated as Figure 40. This code segment indicates that the addition of alternative constructs will affect the knots measure in programs written in FORTRAN-like languages. For languages with an IF...THEN...ELSE control flow operator, the inclusion of an alternative construct does not affect K. Thus K may depend not only on the particular linearization, but also on the implementation language. Another observation motivated by Figure 40 is that, while the knots count in a program written in a FORTRAN-like language is
affected by the inclusion of an alternative construct, the inclusion in such a program of a simple iterative construct will not necessarily affect the knots count. In fact, the knots count is unaffected when arbitrarily nested (structured) iterative constructs are added to simple straight line code. Thus programs composed in this manner have the same zero knots count as any straight line code. The addition of arbitrarily nested (structured) alternative constructs will increase the knots count. This discrepancy of the behavior of K for iterative and alternative constructs is a definite deficiency in a general measure of program control flow complexity.

\[ \text{FORTRAN Implementation of IF...THEN...ELSE with One Knot} \]

Figure 40
With respect to the two structuring transformations, $K$ indicates that they are never inadvisable. This follows from the fact that programs which are well structured will have $K = 0$ (assuming the implementation language supports IF...THEN...ELSE and DO...WHILE). In other words, programs composed using only IF...THEN...ELSE and DO WHILE control flow operators will have control flow arcs which are nested—they do not cross. Therefore, any set of transformations which structures a program must result in a program with a $K$ value no greater than that of the original program. While the analysis of $E$ using these transformations indicates problems because $E$ would dictate these transformations are never advisable, the result for $K$ which indicates they are always advisable should also evoke some skepticism. There is clearly no consensus that the programs produced by these transformations invariably reduce programming difficulty.

A final negative observation on $K$ is based on the fact that any arbitrary structured program has $K = 0$. Therefore, according to $K$, a program with "a large amount" of structured transfer of control will have the same control flow complexity as any simple sequence of straight line code.
4.3. Normal Number

Of the three control flow complexity measures considered, normal number is the least useful for addressing current software engineering issues related to program control flow. The shortcomings of NN as a measure of control flow complexity are:

1. The measure must fail to capture differences in linearizations of a flowgraph since the measure is based on the flowgraph itself. Any two linearizations of the same flowgraph have the same normal number.

2. The normal number measure is applicable to transformations which structure multiple exit loops, but there are instances in which the measure's behavior contradicts accepted notions about structuring loops. The flowgraphs in Figures 34 and 35 provide an example. The flowgraph in Figure 34 with a simple loop with k exits is in normal form with NN = 1. The flowgraph in Figure 35 which results when a boolean variable is used to structure the loop is also in normal form and has NN = k. Thus NN indicates that the structured version is significantly worse. In fact, flowgraphs in normal form were purported to have properties of structured programs, but Figure 34 clearly shows there are normal form flowgraphs that are
not well structured.

3. NN is not applicable to consideration of structuring flowgraphs by node splitting since node splitting may be used to produce the normal form of the flowgraph from which the normal number is determined.

4. The transformation(s) required to produce the normal form of a flowgraph may in themselves obviate the utility of the normal number measure. That a significantly altered flowgraph (in terms of control flow) may have the same measure as the original form of the flowgraph is an undesirable feature of a general measure of program control flow complexity. The node splitting transformations provide a concrete example. Since these transformations are used to derive the normal program form on which NN is based, they leave the measure invariant. But it has already been argued that such transformations can have a decreasing effect on program complexity.

Thus, NN adds little insight into the control flow complexity deficiencies of the software effort measure.
4.4. Cyclomatic Number

Cyclomatic number, as described in Section 2.3, is a measure of the underlying flowgraph of a program which is well founded in graph theory. Unlike normal number, it is not based on a transformed version of the flowgraph. However, the measure as developed in graph theory is not immediately applicable to flowgraphs. The graph theoretic results do not insure that the circuits which make up a basis are directed circuits. This deficiency is described in Section 2.3. The following theorem expresses the desired property for flowgraphs:

Theorem: Let G be a flowgraph with n nodes and e arcs. Add to G the arc from the exit node to the entry node and denote the resulting directed graph by G'. Then there exists

\[ C = e - n + 2 \]

linearly independent directed circuits in G'.

The theorem is proved by using a particular spanning tree and showing that the arcs in G which are not in the spanning tree can be used to construct the desired basis. The
algorithm for generating the desired spanning tree appears in Figure 41. Call the spanning tree produced by the algorithm \( T(G) \) and call the arcs in \( G' \) numbered by the algorithm (those arcs in \( G' \) which are not in \( T(G) \)) "added arcs". Figures 42 and 43 provide an example. In Figure 43 the spanning tree is denoted by solid arrows and the added arcs by dotted arrows. The following Lemmas are used to establish the proof of the preceding theorem:

Procedure \textsc{Depth}(G)

1. Number the arc from the exit node to the entry node "1".
2. Set \textsc{Added-Arc-Number} to 2.
3. Set \textsc{Current-Node} to "entry node".
4. Repeat 4.1 - 4.2 until the value of \textsc{Current-Node} is "entry node" and there are no untraversed arcs out of the entry node.
   4.1. Note that \textsc{Current-Node} has been visited.
   4.2. If the current node has an out arc which has not been traversed then execute 4.2.1 and 4.2.2.
      4.2.1. Traverse one of the untraversed arcs and note that it has been traversed.
      4.2.2. If the node at the end of the arc has not been visited, include the arc just traversed in the spanning tree and set \textsc{Current-Node} to the node at the end of the arc. Otherwise, number the arc with the value in \textsc{Added-Arc-Number} and increment \textsc{Added-Arc-Number} by 1. (The \textsc{Current-Node} remains that of step 4.2.1.)
   Otherwise (referring to 4.2) back up the last arc traversed and consider the initial node of this arc as \textsc{Current-Node}.
5. Return.

Algorithm for Generating a Depth First Spanning Tree

Figure 41
Flowgraph
Figure 42

Spanning Tree and Added Arcs for Flowgraph in Figure 42
Figure 43
Lemma: Let $G$ be a flowgraph with a spanning tree $T(G)$ generated by the algorithm of Figure 41. Then there are $C(G)$ added arcs in $G'$. 

Proof: It is a well-known result that there are $n-1$ arcs in the spanning tree so that there are $e-(n-1)$ added arcs in $G$. But including the arc from the exit node to the entry node this gives $e-n+2$ added arcs in $G'$. Q.E.D.

Lemma: Let $G$ be a flowgraph with spanning tree $T(G)$. Then, by adding the arc from the exit node to the entry node in $T(G)$, a directed circuit is produced.

Proof: The algorithm of Figure 41 must eventually visit the exit node. The first time the exit node is visited the arc to the exit node will be included in the spanning tree. There must be a path in the spanning tree from the entry node to the node at the initial end of this arc. The result follows immediately. Q.E.D.

Lemma: Assume $G$ and $T(G)$ as in the previous lemma. Let $T(G,j)$ denote the flowgraph produced by the addition of the first $j$ added arcs to $T(G)$ ($T(G,0)$ =
The addition of the \( j \)th added arc \( (j = 1, \ldots, C(G)) \) to \( T(G,J-1) \) yields a directed circuit in \( T(G,J) \) which contains the \( j \)th added arc.

**Proof:** (The proof is by induction on the \( k \)th added arc.) The previous lemma establishes that the addition of the arc from the exit node to the entry node in \( T(G) \) produces a directed circuit. Now consider added arc 2. The two circumstances which could cause this arc to not be included in the spanning tree can be divided as follows:

1. Added arc 2 is a "backarc" (2) leading to a node already visited. There are two ways this can arise. The first is if the exit node has not been visited so this arc simply jumps back to a node of the branch being traversed. The second

\[ (2) \text{ A backarc is an arc which produces a program loop. Such arcs can be identified by a depth first search of a flowgraph G with n nodes. In such a search, let a node be numbered when all arcs out of that node have been traversed. Start the numbering with n and decrement for each node numbered. Then an arc from a node numbered i to one numbered j is a backarc if } i > j. \]
way is similar, only it occurs after the exit node has been visited. In both cases a directed circuit is formed.

2. Added arc 2 is not a backarc. In this case the exit node has already been visited and added arc 2 has its terminal node on that path from the entry node to the exit node. Since there must be a path to the initial node of added arc 2 from the entry node and since the terminal node lies on a path from the entry to the exit, then the addition of added arc 2 to T(G) must produce a directed circuit.

A similar argument is applicable to the more general situation. Suppose that k arcs have been added to T(G) and each of the k additions has produced a directed circuit. Then the (k+1)st added arc must either be a backarc or will have its terminal node lying on a path from the entry node to the exit node in T(G,k). In either case, the addition of the (k+1)st added arc must yield a directed circuit. Q.E.D.

These lemmas directly lead to the proof of the theorem.

**Proof:** For an arbitrary flowgraph G construct the spanning tree T(G) by the algorithm in Figure 41. Add
to this spanning tree the added arcs in the specified order. Since the addition of each added arc produces a directed circuit, and since each circuit contains an arc not in any of the previous directed circuits, the new arc produces a directed circuit which is linearly independent of the previous circuits. Thus the addition of the C added arcs yields C linearly independent directed circuits. Q.E.D.

As a result of this theorem the application to flowgraphs of cyclomatic number as a measure of control flow complexity is well founded. The analysis of the measure using the three transformations developed in Section 4.1 proceeds readily. Since cyclomatic number is a measure of a program's underlying flowgraph it is not applicable to the linearization issue. The question of the behavior of cyclomatic number under the two structuring transformations is established in the next two theorems. In order to address the node splitting issue it is helpful to make use of the notion of node splitting which duplicates the node to be split and all the paths from that node to the exit (along with in-arcs of nodes on such paths). In order to avoid the difficulty of excess node duplication depicted by Figures 20 and 21, before any node A is split it will be required that all nodes lying on paths from A to the exit node which are
node splitting candidates (have in-degree greater than one and are not FOLLOW nodes) must be split before A. The procedure STRUCTURE1 in Figure 44 accomplishes this task. When a node A in a flowgraph which is to be split has had all the nodes on paths from A to the exit structured by STRUCTURE1, A is "a viable node splitting candidate".

Procedure STRUCTURE1(N)
1. Let M be an immediate successor of N which has not been structured.
2. Repeat steps 2.1 - 2.2 while M = Ø.
   2.1. Call STRUCTURE1(M).
   2.2. Let M be a successor of N which has not been structured.
   
   /* At this point it is assured that the subgraph emanating from N is such that N dominates every node in that subgraph. */
3. If N has in degree greater than or equal to 2 and N = FOLLOW(P) for an IF node P then execute steps 3.1 - 3.2.
   3.1. For each arc terminating at N generate a unique copy of N and label these nodes N1, ... , Nk.
   3.2. For each Ni produce a unique copy of the subgraph emanating from Ni - the final arcs in the subgraph will terminate in one FOLLOW node.
4. Mark N (or each Ni) as structured.
5. Return.

Call STRUCTURE1(entry node)

Alternate Node Splitting Procedure STRUCTURE1

Figure 44

Theorem: Let G be a flowgraph in which a node D is a viable candidate for node splitting according to the algorithm STRUCTURE1. Also let G' be the flowgraph
produced when \( D \) is split. Then \( C(G') = C(G) \) if and only if the subgraph emanating from \( D \) represents straight line code.

**Proof:** The flowgraph \( G \) may be fairly represented by Figure 45. It may be assumed without loss of generality that the FOLLOW node which occurs at the end of paths from \( D \) is the exit node. Let \( GD \) be the subgraph emanating from \( D \), including the node \( D \), all out-arcs from \( D \), and those arcs which terminate at the exit node (which lie on paths from \( D \)). Then let \( G \) consist of the remainder of the arcs and nodes in \( G \). By denoting the number of arcs by \( e \) and the number of nodes by \( n \), the cyclomatic complexity of \( G \) may be expressed as

\[
C(G) = [(e(G) + e(GD)) - (n(G) + n(GD))] + 2
\]

Figure 46 depicts the result of splitting \( D \). The cyclomatic complexity of this flowgraph \( G' \) is

\[
C(G') = [e(G') + e(GD1) + e(GD2)] - [n(G') + n(GD1) + n(GD2)] + 2
\]

Noting that \( e(G) = e(G') \), \( e(GD) = e(GD1) = e(GD2) \), \( n(G) \)
= \( n(G') \), and \( n(GD) = n(GD1) = n(GD2) \), the cyclomatic complexity of \( G' \) may be expressed as

\[
C(G') = [e(G) + 2e(GD)] - [n(G) + 2n(GD)] + 2
\]

Thus \( C(G') = C(G) \) if and only if

\[
e(GD) - n(GD) = 0
\]

The only way for the number of arcs in \( GD \) to be equal to the number of nodes in \( GD \) is if \( GD \) represents straight line code. Q.E.D.
D as Node Splitting Candidate
Figure 45

The Result after D is Split in Flowgraph in Figure 45
Figure 46
**Theorem:** Let $G$ be a flowgraph which contains a loop with $k$ exits and let $C(G)$ be the cyclomatic complexity of $G$. Then the cyclomatic complexity of the flowgraph $G'$ produced by structuring the loop in $G$ using a boolean variable is

$$C(G') = C(G) + 1$$

**Proof:** The flowgraph $G$ and the structured flowgraph $G'$ are depicted in Figures 47 and 48. The cyclomatic complexity of $G$ may be expressed as

$$C(G) = [e(G) + e(G')] - [n(G) + n(G')] + 2$$

and the cyclomatic complexity of $G'$ as

$$C(G') = [e(G') + e(G')] - [n(G') + n(G')] + 2$$

The following relationships are used to express $C(G')$ in terms of $e(G)$, $e(G')$, $n(G)$, and $n(G')$:

- $e(G') = e(G)$
- $e(G') = e(G) + k + 3$
- $n(G') = n(G)$
- $n(G') = n(G) + k + 2$
Thus

\[ C(G') = (e(G) + e(G_l) + k + 3) - \]
\[ [n(G) + n(G_l) + k + 2] + 2 \]
\[ = [e(G) + e(G_l)] - [n(G) + n(G_l)] + 3 \]
\[ = C(G) + 1 \quad Q.E.D. \]

**Multiple Exit Loop**

*Figure 47*
The implication of the two theorems for cyclomatic complexity is that the measure's behavior is not out of line with current debate on these different program forms. In the case of the node splitting transformation, which has wider acceptance, C indicates that control flow complexity may not be adversely affected. For the case of structuring multiple exit loops, the small increase in control flow complexity indicated by C is not unreasonable.
With the analyses of the three measures of control flow complexity which are presented in this chapter, it is clear that cyclomatic complexity is the most appropriately sensitive of the measures. This sensitivity, coupled with the measure's firm graph theoretic grounding, makes the control flow complexity measure \( C \) the measure which is most likely to provide insight into improving the behavior of the more general program complexity measure \( E \). The control flow complexity measures alone are inadequate as general measures of control flow complexity because they fail to capture any complexity incurred within predicates or statements. \( E \) has been shown to be an adequate measure of this straight line code complexity. This possibility of using \( C \) to improve \( E \) is given even greater support in light of the fact that the one notion of control flow complexity to which \( C \) is not appropriately sensitive, issues of linearization, is captured appropriately by \( E \).

In the consideration of \( E \) and \( C \) which is presented in Chapter 5, two additional properties of cyclomatic complexity are used. These properties are developed in the following theorems:

**Theorem**: Let \( G_1 \) and \( G_2 \) be flowgraphs with cyclomatic complexities \( C(G_1) \) and \( C(G_2) \), respectively. Then the cyclomatic complexity of the flowgraph \( G \) representing
the alternative construct

\[
\text{IF (P) THEN BEGIN G1 END} \\
\text{ELSE BEGIN G2 END}
\]

is

\[
C(G) = C(G1) + C(G2)
\]

**Proof**: Let \( n \) represent number of nodes and \( e \) number of arcs, as before. The flowgraph representation \( G \) of the alternative construct appears as Figure 49. Using the formula for cyclomatic complexity yields

\[
C(G) = (e(G1) + e(G2) + 4) - \\
(n(G1) + n(G2) + 2) + 2 \\
= (e(G1) - n(G1) + 2) + \\
(e(G2) - n(G2) + 2) \\
= C(G1) + C(G2) \quad \text{Q.E.D.}
\]
Theorem: Let G₁ be a flowgraph with cyclomatic complexity C(G₁). Then the cyclomatic complexity of the flowgraph G corresponding to the iterative program construct

\[ \text{WHILE } \text{(P)} \text{ DO } \]

\[ \text{BEGIN G₁ END} \]

is

\[ C(G) = C(G₁)+1 \]

Proof: Again let \( n \) represent number of nodes and \( e \) number of arcs. The flowgraph representation G of the iterative construct appears as Figure 50. From this figure the cyclomatic complexity of G can be determined.
from the definition:

\[ C(G) = (e(G1) + 3) - (n(G1) + 2) + 2. \]
\[ = (e(G1) - n(G1) + 2) + 1 \]
\[ = C(G1) + 1 \quad \text{Q.E.D.} \]

Flowgraph Representation of an Iterative Construct

Figure 50
CHAPTER 5.

A SYNTHESIZED PROGRAM COMPLEXITY MEASURE

The analyses of Chapter 3, along with the evaluations by other researchers presented in Chapter 2, provide strong evidence that $E$ is a good program complexity measure. The one negative result in all the evaluations of $E$ is that this measure fails to capture certain aspects of program control flow complexity. Three measures which purport to quantify this aspect of program complexity were analyzed in Chapter 4. Cyclomatic complexity is shown to be the most appropriately sensitive to currently discussed issues of control flow complexity. The one issue on which $C$ fails to behave correctly is that of linearization. However, this is the one control flow complexity issue which $E$ captures appropriately. It would seem that the obvious question motivated by these studies is, "Can $E$ and $C$ be synthesized into a new program complexity measure which overcomes the weaknesses of either $E$ or $C$ taken alone?" This Chapter discusses of the feasibility of successfully accomplishing
this synthesis.

5.1. A First Attempt

Cyclomatic complexity is the maximum number of linearly independent circuits in a flowgraph, i.e. the number of circuits in a basis for that flowgraph. The demonstration in Chapter 4 that a basis of directed circuits can always be determined for a given flowgraph contains a procedure for producing such a basis. One approach to synthesizing E and C is to perform the software science count of operator and operand occurrences along each circuit in the basis, avoiding duplication of counts for the node which appears twice in each circuit. Then these occurrences for each circuit can be added and the sums used in computing effort. More formally:

**Definition**: Let P be a program with corresponding flowgraph G and let L1, L2, ..., LC be the circuits in a basis for G determined by the aforementioned procedure. Denote the operator and operand occurrence counts for the circuit Li as N1(Li) and N2(Li), respectively (the repeated node is not "counted" twice). Then the operator and operand occurrence counts for the entire program can be taken to be:
\[ N_1(PCI) = N_1(Li) \]
\[ N_2(PCI) = N_2(Li) \]

Then these occurrence counts can be used in a modified volume measure

\[ V(PCI) = (N_1(PCI) + N_2(PCI)) \log (n_1 + n_2) \]

This modified volume measure, obtained from the basis which yields the measure \( C \), is used in a new measure of program complexity \( PCI \):

\[ PCI = \frac{(V(PCI) \uparrow 2)}{V^*} \]

Despite its intuitive appeal, \( PCI \) is not well defined. There are two shortcomings and both stem from the procedure used in the demonstration that a basis of directed circuits can always be determined for any flowgraph (the directed arc procedure). (These deficiencies in no way detract from the procedure's original use in the proof.) The first problem is that given a particular spanning tree, a basis is not completely specified. Consider the flowgraph in Figure 51 along with a spanning tree and added arcs for that flowgraph which appear in Figure 52. The first two circuits in the basis are explicitly determined, but the second two are
not. The ambiguity is a result of the fact that after the first two added arcs are added to the spanning tree, there is a choice of how to get from node F to the exit node. Using the symbol "v" to denote alternative paths in a circuit, the bases from this spanning tree are:

Entry  A  B  F  H  I  Exit  Entry
Entry  A  B  F  G  I  Exit  Entry
Entry  A  C  D  F  [H  v  G]  I  Exit  Entry
Entry  A  C  E  F  [H  v  G]  I  Exit  Entry

This problem for PCI has a possible resolution. Such choices can be encountered only in situations similar to the one depicted in Figures 51 and 52. Generally, the choices will occur when, in order to complete a circuit for a given added arc, it is necessary to traverse a section of the modified spanning tree where all of the paths have already been added. (In fact, such choices can occur only when all the paths of a section of the flowgraph have been traversed. To see this suppose that in order to complete a circuit for added arc k there is a choice of paths between added arc k and the exit node, and that at least one of the choices is untraversed. Then there is an untraversed path which must contain an added arc which is "deeper" (referring to the depth first search approach of the added arc
Spanning Tree and Added Arcs for Flowgraph in Figure 51

Figure 52
algorithm) than added arc $k$. Therefore this arc will be numbered $j$ where $j > k$, since it is untraversed. This contradicts the fact that the added arc algorithm would insure $j < k$.) When the operator and operand occurrence counts are determined for a circuit which contains this type of choice, the average occurrence counts of the paths in the section with alternatives can be used. Since all the paths in that section must appear in other circuits of the basis, this approach may not be unreasonable.

However, the second drawback to PCI demonstrates a more serious flaw in the measure. The directed arc procedure does not even generate a unique spanning tree for a given flowgraph, and distinct spanning trees yield different bases. Clearly, different bases yield different PCI values. Figures 53 and 54 provide an example. Both Figures 53 and 54 are spanning trees with added arcs for the same flowgraph. The directed circuit procedure could generate either spanning tree. This is a result of the fact that the procedure is nondeterministic in its choice of untraversed arcs out of the current node. In Figure 53 the arc AB is traversed before the arc AC. Figure 54 is produced when the arc AC is traversed first. The bases (that there are more than one stems from the first problem discussed) resulting from Figure 53 are:
The bases of Figure 54 are:

Entry C E F Exit Entry
Entry C E Exit Entry
Entry C I G Exit Entry
Entry C I F Exit Entry
Entry B D F Exit Entry
Entry B E [F Exit Entry v Exit Entry]

The problem with having the two different basis sets is that one will yield a PCI value which "weights" operator and operand occurrences at a given node differently than some other basis set will weight that same node. For instance, any basis produced from Figure 53 contains three occurrences of the node B while any basis generated by Figure 54 contains two. This discrepancy could be overcome by computing PCI for each spanning tree (using the averaging over path choices within each spanning tree already
One Spanning Tree for a Flowgraph

Figure 53

Another Spanning Tree for the Flowgraph in Figure 53

Figure 54
This approach is at least well defined since for any given flowgraph there are only a finite number of nondeterministic choices which are made by the directed arc procedure. These PCI values could then be averaged to obtain a program complexity measure. Even if this approach were not so cumbersome (a very real, practical limitation), it fails to solve a very fundamental problem with this particular method of synthesizing effort and cyclomatic complexity. The node C in the flowgraph of Figures 53 and 54 is going to influence this average of the PCI values to a greater extent than is the node B. (The basis sets given are the only two for this flowgraph. Different spanning trees do not always produce different bases.) The discrepancy is the result of the greater number of paths from node C to the exit node. However, following this approach yields undesirable behavior of the proposed measure since the addition of arbitrarily complex structured constructs on a path from B to the exit node would radically alter the relative complexities of B and C. This results from the underlying problem that B's contribution to effort grows linearly with each new path from B. It would seem that once the predicate of node B is written that the actions on subsequent paths from B should not weigh so heavily in a final measure of the effort involved in writing that predicate.
5.2. Limiting the Domain to Define PC

These two problems with the directed arc procedure and the subsequent weighting problem just discussed indicate that the approach taken to synthesizing PCI from E and C for general flowgraphs has inherent limitations. A possible strategy for eliminating these limitations is to restrict the nature of the programs and flowgraphs to which the measure is to be applied. A first attempt at such a restriction is to consider structured programs. Recall that structured programs are constructed from D-structures using composition and sequencing of those single entry, single exit constructs. However, the same problems which plagued the applicability of PCI to arbitrary programs and flowgraphs still pertain to this restricted situation. This can be seen in the flowgraph of Figure 51. There are different spanning trees for this flowgraph, despite the fact that it is structured.

In fact, the flowgraph of Figure 51 suggests that it is the sequencing of structured constructs which leads to the problems with having different spanning trees which produce different bases for a flowgraph. This conjecture is verified by the proof of the following theorem:

**Theorem**: Let G be an arbitrary flowgraph constructed
from flowgraph constructs corresponding to D-structures and composition (nesting) of these constructs (no sequencing is allowed). Call flowgraphs constructed in this manner "nested structured constructs". Then there is only one basis for the flowgraph G.

Proof: Figure 55 reveals that there is only one possible spanning tree (which can result from a depth first search) for the basic iterative D-structure. Figures 56 and 57 reveal that there are two possible spanning trees for the alternative D-structure, but both of these spanning trees yield the same basis. Now suppose an arbitrary D-structure is substituted for node B in the basic iterative structure. Figure 58 reveals that the substitution of an iterative structure does not affect the unique basis result. The basis generated must be

Entry A Exit Entry
A' B' A'
A A' A

Figure 59 demonstrates the same result for substitution of an alternative construct for node B in the basic iterative structure. The basis generated must be
A similar result is obtained when basic D-structures are substituted for nodes B and C of the basic alternative structure. The iterative and alternative structures are substituted for node B of the basic alternative construct of Figure 56 in Figures 60 and 61, respectively. The basis from Figure 60 must be:

Entry A D Exit Entry
D E D
Entry A C Exit Entry

and the basis from Figure 61 must be:

Entry A D E Exit Entry
Entry A D F Exit Entry
Entry A C Exit Entry

This basis is generated regardless of whether the spanning tree with the arc "E Exit" or the spanning tree with the arc "F Exit" is used. Similarly, the basis associated with Figure 60 and the one associated
with Figure 6! would be obtained (isomorphically) if, instead of substituting for node B in the basic alternative construct, the node C was replaced. In summary, the nesting of a structured construct in a structured construct does not alter the fact that the resulting composite structure has a single basis.

Q.E.D.

ENTRY

A

\( \downarrow \)

B

EXIT

Spanning Tree for the Iterative Construct

Figure 55
One Spanning Tree for the Alternative Construct

Figure 56

The Second Spanning Tree for the Alternative Construct

Figure 57
Spanning Tree for an Iterative Structure Nested in an Iterative Structure

Figure 56

Spanning Tree for an Alternative Structure Nested in an Iterative Structure

Figure 59
Spanning Tree for an Iterative Structure Nested in an Alternative Structure

Figure 60

Spanning Tree for an Alternative Structure Nested in an Alternative Structure

Figure 61
Consideration of the bases presented in the proof of the preceding theorem reveals that each element of these bases is an elementary cycle. An elementary cycle is one in which only the first node of the cycle is repeated. It is clear that there are elementary cycles in an arbitrary flowgraph which are not necessarily part of a particular basis for that flowgraph (e.g. see Figure 53 and its bases). However, one might conjecture that the elementary cycles of a flowgraph constructed by only nesting D-structures constitute a basis of that flowgraph. The number of elementary cycles in such a flowgraph may be determined recursively by:

1. For the iterative construct

   WHILE A DO G1;

   The number of elementary cycles is one plus the number of elementary cycles of the form A G1 A, where there may be elementary cycles contained strictly in G1.

2. For the alternative construct

   IF A THEN BEGIN G1 END;
   ELSE BEGIN G2 END;

   The number of elementary cycles is the number of
elementary cycles of the form $A G_1 A$ plus the number of elementary cycles of the form $A G_2 A$. Again, there may be elementary cycles contained entirely in $G_2$.

3. For simple straight line code

$$A;$$

The number of elementary cycles is taken to be one, corresponding to the one circuit Entry $A$ Exit Entry in the case of such a simple program.

Abbreviate number of elementary cycles by NEC. The number of elementary cycles in the flowgraph in Figure 62 is determined by the definition as follows:

$$\text{NEC}(G) = \text{NEC of the form } A G_1 A +$$
$$\text{NEC of the form } A G_5 A$$
$$= [1 + \text{NEC of the form } B G_2 B] +$$
$$[\text{NEC of the form } E J E + \text{NEC of the form } E G_6 E]$$
$$= [1 + [\text{NEC of the form } C G_3 C +$$
$$\text{NEC of the form } C G_4 C]] +$$
$$[1 + \text{NEC of the form } K L K +$$
$$\text{NEC of the form } K M K]$$
$$= [1 + [1 + \text{NEC of the form } D F D] +$$
$$[\text{NEC of the form } G H G +$$
$$\text{NEC of the form } G I G]] +$$
This recursive procedure for determining the number of elementary cycles in a flowgraph constructed by nesting D-structures is identical to a procedure for determining the cyclomatic number of such structures. This latter procedure is a direct result of the last two theorems proved in Chapter 4. This discussion proves:
Theorem: The cyclomatic complexity of a flowgraph $G$ constructed by composing $D$-structures (no sequencing) is exactly the number of elementary cycles in $G$.

All of this motivates the application of the same approach which was used to synthesize $PCI$ from $E$ and $C$ to a well-defined program complexity measure for arbitrary structured flowgraphs. This measure, $PC$, is based on the notion that structured programs are written by constructing components and then sequencing them. This would imply some sort of independence of sequenced structured constructs. This underlying assumption is motivated by the approach of strict structured programming (e.g. [DIJ76]). Since the synthesis of $E$ and $C$ is well defined for structured programs without sequencing, the operator and operand occurrences can be summed along each elementary circuit in the unique basis for that construct. (These are the elementary circuits of that construct.) Then these sums for each construct can be summed for sequences of such constructs. More formally

Definition: Let $N_1(P)$ indicate the number of operator occurrences in a program segment $P$ and let $N_2(P)$ indicate the number of operand occurrences. Then let $TN_1$ and $TN_2$ represent operator and operand occurrences summed along the elementary circuits of a program
segment P, where tokens in the repeated node are counted in only the first occurrence of that node in the circuit. Then

1. For the iterative construct

\[
\text{WHILE A DO G1}
\]

(where G1 is structured without sequencing) the sum of operator and operand occurrences along the elementary circuits in a program with this form may be determined by:

\[
TN1 = N1(A) + TN1 \text{ (along elementary circuits of the form } A \text{ G1 A) (Recall that there may be elementary circuits contained strictly in G1.)}
\]

\[
TN2 = N2(A) + TN2 \text{ (along elementary circuits of the form } A \text{ G1 A) }
\]

2. For the alternative construct

\[
\text{IF A THEN BEGIN G1 END;}
\text{ELSE BEGIN G2 END;}
\]

(where G1 and G2 are structured without
sequencing) the sum of operator and operand occurrences along the elementary circuits in a program with this form may be determined by:

\[ TN_1 = TN_1 (\text{along elementary circuits of the form } A \ G_1 A) + TN_1 \]
\[ TN_2 = TN_2 (\text{along elementary circuits of the form } A \ G_1 A) + TN_2 \]

3. For the straight line code segment

\[ A \]

\[ TN_1 = N_1(A) \]
\[ TN_2 = N_2(A) \]

4. For sequences of two nested structured constructs \( G_1 \) and \( G_2 \) the token counts can be determined by

\[ TN_1 = TN_1(G_1) + TN_1(G_2) \]
\[ TN_2 = TN_2(G_1) + TN_2(G_2) \]
5. For the alternative construct

\[
\text{IF A THEN BEGIN G1 END;}
\]
\[
\text{ELSE BEGIN G2 END;}
\]

where G1 and G2 are sequences of structured constructs then

\[
T_{N1} = (N_1(A) + T_{N1}(G_1)) + (N_1(A) + T_{N1}(G_2))
\]
\[
T_{N2} = (N_2(A) + T_{N2}(G_1)) + (N_2(A) + T_{N2}(G_2))
\]

6. For the iterative construct

\[
\text{WHILE A DO G1;}
\]

where G1 is made up of sequences of structured constructs

\[
T_{N1} = N_1(A) + (N_1(A) + T_{N1}(G_1))
\]
\[
T_{N2} = N_2(A) + (N_2(A) + T_{N2}(G_1))
\]

These operator and operand occurrence counts TN1 and TN2 are used in a synthesized program complexity measure PC as follows:
\[ V(\text{PC}) = (T_{N1} + T_{N2}) \log (n1 + n2) \]

\[ PC = \frac{(V(\text{PC}) \uparrow 2)}{V*} \]

To summarize the development of this definition, the approach to synthesizing \( E \) and \( C \) in a program complexity measure by performing operator and operand occurrence counts along the circuits in a basis of a flowgraph is actually not well defined. The problems with this approach are attributable to the fact that general flowgraphs may have different bases. By limiting consideration to structured flowgraphs constructed by nesting D-structures, the problem of multiple bases is eliminated. Furthermore, the circuits of the unique bases of such flowgraphs are exactly the set of elementary cycles in the flowgraph. Therefore, the token counts are being performed along each "basic path" in a flowgraph. Then, with the assumption that sequences of these nested structured constructs are designed by writing each nested construct (reasonably) independently, overall token counts may be determined by summing the counts for the nested constructs.
5.3. An Evaluation of PC

The analytical evaluation of PC is quite simple. The measure certainly takes care of the underlying objection to the software effort measure E. E is defined so that "+" influences program complexity in the same way that "IF" does. This is not true for PC since "IF" contributes to the number of elementary circuits in a nested structured construct. In addition, PC must behave exactly as C when viewed in light of node splitting transformations and hence is an improvement over E. This result is an immediate consequence of the fact that node splitting does not change the possible paths through the program. PC is superior to C as a general measure of program complexity, since C only measures control flow complexity and so cannot account for complexity incurred by straight line code. Recall that Hansen was the first to work on this limitation of C [HAN78].

Two transformations used in the analytical evaluation of the previous measures are not applicable to PC because it is only defined for structured constructs. (This is also the case for node splitting, adhering strictly to the definition of PC. However, the extension of PC in node splitting in lattice-like flowgraphs seems clear because the circuits in the basis are not altered.) Since differences
in linearizations must lie in the different use of unrestricted transfer of control, so that such linearizations cannot be structured, PC is not applicable. Similarly, PC is not defined for multiple exit loops so that analysis by structuring such loops is not possible.

The following theorem facilitates the computation of PC:

**Theorem:** In a structured program P let

- \( p \): the token counts appearing in the conditions (simple or compound predicates) of P
- \( r \): the token counts of the remainder of the program P

Then the program complexity measure PC is determined by computing \( V(\text{PC}) \) as:

\[
V(\text{PC}) = (2p + r) \log (n1 + n2)
\]

**Proof:** This is actually an immediate consequence of the definition. For the iterative nested structured constructs the paths along which TN1 and TN2 are counted are of the form "Entry predicate Exit" and "predicate loop body". For the alternative nested
structure the paths are of the form "predicate 'then' clause" and "predicate 'else' clause". Thus the result for nested structured constructs is clear. Since sequencing simply involves summing the counts of nested structured constructs, the general result follows immediately. Q.E.D.

Ironically, this brings the analysis of measures of program complexity "full-circle". Most of the work up to the time of this research has been empirical. An analytical approach to such analyses has yielded a new measure which ought now to be subjected to empirical evaluation. To this end, a portion of Gordon's empirical study of E which used pairs of programs consisting of a poor program and an improved program is repeated on PC. The pairs used are from the paper by Gileadi and Ledgard [GIL74] and the paper by Kernighan and Plauger [KER74]. These sources are used because they represent a wide cross section of examples, and did not restrict consideration to one aspect of program complexity.

The results of this empirical study are presented in Table 6. The program pairs and token counts appear in Appendix A. The "poor" version ("poor" at least in the opinion of the original author) of each program is the "A" version. Some of the pairs of programs which appear in the
original sources cannot be used to evaluate PC since they are not structured programs. In the eleven program pairs from these papers to which PC is applicable, E' fails to work in two cases. In one of these cases, PC reflects the appropriate reduction. In the one case in which PC fails, the "poor" version makes use of the properties of integer arithmetic in a rather obscure way. Whether this is actually a poor program is probably dependent on the fluency of the judge using languages with such facilities, however the corresponding version which does not use this facility will obviously have a greater PC measure. Therefore, in this admittedly preliminary empirical study, PC is an improvement over E'.

In summary, the answer to the question of whether or not the drawbacks to software effort and cyclomatic complexity might be eliminated in a synthesized program complexity measure is affirmative. Admittedly, PC must somehow be made more general to account for unstructured programs, but the results presented here certainly indicate that synthesizing E and C may well prove to be a fruitful avenue of research.
Table 6
Comparison of Performance of PC and E'

<table>
<thead>
<tr>
<th>Example</th>
<th>Gordon's Number</th>
<th>PC</th>
<th>E'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2A</td>
<td>7889</td>
<td>5518</td>
</tr>
<tr>
<td></td>
<td>2B</td>
<td>7136+</td>
<td>3460+</td>
</tr>
<tr>
<td>2</td>
<td>4A</td>
<td>6418</td>
<td>979</td>
</tr>
<tr>
<td></td>
<td>4B</td>
<td>8450+</td>
<td>1607+</td>
</tr>
<tr>
<td>3</td>
<td>3A</td>
<td>6418</td>
<td>979</td>
</tr>
<tr>
<td></td>
<td>3B</td>
<td>5887+</td>
<td>483+</td>
</tr>
<tr>
<td>4</td>
<td>6A</td>
<td>995</td>
<td>2008</td>
</tr>
<tr>
<td></td>
<td>6B</td>
<td>393+</td>
<td>483+</td>
</tr>
<tr>
<td>5</td>
<td>8A</td>
<td>393</td>
<td>483</td>
</tr>
<tr>
<td></td>
<td>8B</td>
<td>65+</td>
<td>54+</td>
</tr>
<tr>
<td>6</td>
<td>9A</td>
<td>3839</td>
<td>1869</td>
</tr>
<tr>
<td></td>
<td>9B</td>
<td>2086+</td>
<td>1020+</td>
</tr>
<tr>
<td>7</td>
<td>10A</td>
<td>63637</td>
<td>7304</td>
</tr>
<tr>
<td></td>
<td>10B</td>
<td>11905+</td>
<td>1525+</td>
</tr>
<tr>
<td>8</td>
<td>11A</td>
<td>18798</td>
<td>6263</td>
</tr>
<tr>
<td></td>
<td>11B</td>
<td>14112+</td>
<td>4833+</td>
</tr>
<tr>
<td>9</td>
<td>14A</td>
<td>2756</td>
<td>980</td>
</tr>
<tr>
<td></td>
<td>14B</td>
<td>2613+</td>
<td>1062+</td>
</tr>
<tr>
<td>10</td>
<td>15A</td>
<td>794</td>
<td>1186</td>
</tr>
<tr>
<td></td>
<td>15B</td>
<td>393+</td>
<td>483+</td>
</tr>
<tr>
<td>11</td>
<td>18A</td>
<td>&gt;94,677 (1)</td>
<td>94,677</td>
</tr>
<tr>
<td></td>
<td>18B</td>
<td>51,609+</td>
<td>27,246+</td>
</tr>
</tbody>
</table>

(1) In this program it is clear that any method of counting operators and operand occurrences which reflects control flow complexity by counting some tokens more than once, must yield a value for PC which is greater than that for E'.

---
CHAPTER 6

SUMMARY: THE OBJECTIVE IS CLOSER

The software crisis is a direct result of the intellectual unmanageability of many existing software systems. This crisis is forcing increased awareness of the fact that decreasing programming difficulty, the mental effort expended by a programmer when she or he produces or maintains a given software segment, is an imperative objective for computer scientists. It is equally important that methods for decreasing programming difficulty be evaluated in a sound scientific manner. This scientific approach to easing the software crisis is an important component of the growing area of software engineering.

More specifically, the crisis can be eased by demonstrating empirically and analytically that particular language features and programming methodologies reduce programming difficulty. Most of the work which has been concerned with this type of evaluation has been empirical in nature. Section 2.1 contains a review of attempts to
compare the use of various features and methodologies with measures of programming difficulty. This summary reveals that there are (at least) practical limitations to this approach to evaluation of software engineering principles. These limitations provide considerable motivation for a measure, derivable from actual source programs, which quantifies the programming difficulty which is incurred by interacting with those programs. Any such measure is called a program complexity measure.

This perspective on the motivation and nature of measures of program complexity is the first contribution of this particular research effort toward achieving the goal of establishing an adequate measure. After this initial statement of the nature of the problem, an analytical methodology for evaluating program complexity measures is formalized. This is done not to imply that empirical studies to evaluate these measures are unimportant, but rather to fill a void in existing approaches. Much of the development of empirical evaluations of software which resulted from studies of particular language features and programming methodologies is applicable in evaluations of program complexity measures.
The formalized analytical methodology is first applied using principles of program modularity to evaluate the software science effort measure \( E \). In order to apply the particular transformations used in this analysis, principles of data flow analysis are used to carefully define the environments in which repeated similar code sequences may be reduced. In the case of reducing these repetitions by defining a subprogram, the data flow analysis leads to formalized results on issues of parameterization and global versus local variables. The behavior of \( E \) in these modularizing circumstances indicates that \( E \) is an adequate program complexity measure.

The analytical methodology is next applied using principles of program control flow. The entire structured programming issue is integrally concerned with control flow so that, clearly, this is an important factor which a general measure must capture. Two of the control flow issues used in the analytical evaluations are not specifically developed in the open literature. Therefore, the notion of a linearization of a flowgraph is defined, and the various meanings of "node splitting" are discussed. A node splitting procedure which is nondeterministic with respect to the order in which nodes are split is presented. This nondeterminism is shown to be immaterial. The software effort measure is then evaluated using transformations which
embody the control flow issues of linearization, node splitting, and structuring multiple exit loops. The evaluation reveals a weakness in E pertaining to control flow complexity since E behaves aberrantly under node splitting and structuring multiple exit loops. It is postulated that this lack of sensitivity of E to control flow complexity may be the result of the fact that in the quantification of E an occurrence an operator which occurs in expressions has the same influence as an occurrence of a control flow operator.

Since E is wanting as a measure of control flow complexity, existing attempts to define adequate control flow measures are evaluated using the same three transformations. It is shown that there are serious deficiencies in the knots measure (K) and in the normal number measure (NN). A more promising measure is cyclomatic complexity (C). It is shown that this graph theoretic measure is applicable as a measure of program control flow complexity, in that the bases of circuits from which the measure is determined can always be composed of only directed circuits. The behavior of C under node splitting and structuring multiple exit loops reveals that in both these instances C's behavior is in line with currently accepted notions concerning these transformations. However, C is incapable of distinguishing between linearizations of
the same flowgraph. In a final contribution to the understanding of C, two properties which relate C, and the construction of structured programs are established.

Since cyclomatic complexity captures those notions of control flow complexity which E misses, the question of the feasibility of synthesizing these two measures in an improved program complexity measure is addressed. It is shown that a unique basis of circuits does not exist in the cases of arbitrary and structured flowgraphs. However, it is the sequencing of structured constructs which poses the problem of not having a unique basis. It is also shown that a unique basis does exist for any flowgraph constructed of an arbitrary nesting of structured constructs, and that this unique basis is composed of precisely the elementary circuits in such a flowgraph. This lends intuitive appeal to developing an effort like measure in which occurrence counts are determined along each circuit in the basis for nested structured constructs. A theorem which greatly facilitates the computation of this new program complexity measure PC is proved.

Finally, PC is evaluated analytically and empirically. The empirical study compares the behavior of PC with that of software effort and, in this preliminary study, PC is shown to be superior. Thus it would seem that continued work on a
program complexity measure based on a synthesis of E and C is well motivated.

This work leaves several avenues of research untraveled. The first two actually originate with software science studies. Software-science-like measures are based on counting operators and operands in programs. However, no generally accepted schema for classifying operators and operands in an arbitrary programming language currently exists. Since it has been shown that the choice of counting strategy can influence certain software science measures, the need for such a schema is quite clear. (A slightly different approach to substantiating a particular counting strategy is the indirect result of the evaluation of E using modularity principles.) Another unresolved problem in software science stems from the various approaches which have been used to quantify program level. The original definition of level has intuitive appeal, but the way in which minimum volume is quantified may be deficient. A good quantification of minimum volume, possibly using program specifications, might well resolve the level issue. The implication of V* quantification for measures based on software effort is clear from the definition of E.
The analytical evaluation of the four measures of interest in this research is, in all likelihood, not exhaustive. Other transformations may be identified which can be applied with the analytical methodology used in this study, and other analytical methodologies may well be developed. Since this represents an early effort to employ analytical methodologies to the evaluation of program complexity measures, it is at least prudent to assume that the technique can be extended. For instance, other general classes of programming principles might be utilized. A more general notion of modularity provides a specific example.

Finally, PC should be viewed as no more than a positive response to the question of whether E and C might be synthesized in a new program complexity measure. The analytical and empirical evaluation of E is limited and only lends some confidence that continued evaluation is warranted. In continued studies, as this work clearly demonstrates, the measure of interest may well be refined. Specifically, such studies should include attempts to extend PC to arbitrary structured programs. This extension will facilitate the needed continued empirical evaluation of the measure by permitting the use of existing extensive data sets from previous evaluations of other measures. However, despite these needs for continued research, it is reasonable to contend that the research effort summarized in this
dissertation contributes to the goal of a solid program complexity measure.
BIBLIOGRAPHY


APPENDIX A

The program pairs used to evaluate empirically the synthesized program complexity measure PC are enumerated in this Appendix. The basic counts (as tabulated according to the criteria established in the definition for PC) are included after each program, as are the pertinent intermediate measures used to compute PC. The eleven pairs are numbered and Gordon's original number for each program is also included.

1. (2A) The PL/I programs in example 1 are direct implementations of the flowcharts which appear in the original source. All iterative structures in these examples are expressed as WHILE loops.

```plaintext
FOUND = false;
I = 1;
DO WHILE (¬FOUND);
  IF I > M THEN
    DO;
      FOUND = true;
      LIST (I) = X;
      COUNT (I) = 0;
      M = I;
    END;
  ELSE
    IF LIST (I) = X THEN FOUND = true;
    ELSE I = I + 1;
  END;
COUNT (I) = COUNT (I) + 1;
I/O operands: M, LIST, X
```
operatorsoperands
= (assignment) 9FOUND 5
; 13false 1
WHILE 1I 10
grouping 31 3
¬ 2M 2
IF 2true 2
> 1LIST 2
subscript 5X 2
= (relational) 1COUNT 3
+ 20 1

n1 = 10
n2 = 10
TN1 = 39
TN2 = 31

(2B) I = 1;
DO WHILE (I < M & LIST(I) = X);
I = I + 1;
END;
IF I <= M THEN
DO;
LIST(I) = x;
COUNT(I) = 0;
M = I;
END;
COUNT(I) = COUNT(I) +1;

I/O operands: M, LIST, X
operatorsoperands
= (assignment) 6I 14
; 101 3
WHILE 1M 5
grouping 3LIST 3
¬ 2X 3
& 2COUNT 3
subscript 6
¬ 20 1
+
IF 1
<= 2

n1 = 11
n2 = 7
TN1 = 37
TN2 = 32

n2* = 3
V* = 11.6
V(PC) = 302.5
PC = 7888.5

n2 = 10
V(PC) = 287.7
PC = 7135.5
2. (4A) All of the programs in this Appendix are expressed in PL/I.

\begin{verbatim}
I = 1;
DO WHILE (I <= N);
  I = 1;
  DO WHILE (J <= N);
    X (I, J) = (I/J) * (J/I);
    J = J + 1;
  END;
  I = I + 1;
END;
I/O operands: X, N
\end{verbatim}

\begin{verbatim}
operators
= (assignment) 5
; 9
WHILE 2
grouping 6
<= 4
subscript 1
, 1
/ 2
* 1
+ 2
\end{verbatim}

\begin{verbatim}
n1 = 10
n2 = 5
TN1 = 33
TN2 = 25
\end{verbatim}

(4B) I = 1;
DO WHILE (I <= N);
  I = 1;
  DO WHILE (J <= N);
    IF (I = J) THEN X (I, J) = 1.0;
    ELSE X (I, J) = 0.0;
    J = J + 1;
  END;
  I = I + 1;
END:
I/O operands: X, N

\begin{verbatim}
n2* = 2
V* = 8.0
V(PC) = 226.6
PC = 6418.4
\end{verbatim}
3.  \[(3A)\] Same as 4A.

\[(3B)\] \[I = 1;\]
\[\text{DO WHILE (} I \leq N);\]
\[\quad J = 1;\]
\[\quad \text{DO WHILE (} J \leq N);\]
\[\quad \quad X (I, J) = 0.0;\]
\[\quad \quad J = J + 1;\]
\[\quad \text{END;}\]
\[\quad X (I, I) = 1.0;\]
\[\quad I = I + 1;\]
\[\text{END;}\]

I/O operands: \[X, N\]

\begin{tabular}{ll}
\textbf{operators} & \textbf{operands} \\
\text{= (assignment)} & 6 \\
\text{;} & 10 \\
\text{WHILE} & 2 \\
\text{grouping} & 4 \\
\text{<=} & 4 \\
\text{subscript} & 2 \\
\text{,} & 2 \\
\text{+} & 2 \\
\end{tabular}

\[n_1 = 8\]
\[n_2 = 6\]
\[T_{N1} = 32\]
\[T_{N2} = 25\]

\[n_2^* = 2\]
\[V^* = 8.0\]
\[V(\text{PC}) = 217\]
\[\text{PC} = 5886.7\]
4. (6A) The following PL/I implementation uses the IF...THEN...ELSE alternative construct.

```pli
IF X < Y THEN
   IF X < Z THEN SMALL = X;
   ELSE SMALL = Z;
ELSE
   IF Y < Z THEN SMALL = Y;
   ELSE SMALL = Z;
```

I/O operands: X, Y, Z, SMALL

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (assignment)</td>
<td>I  9</td>
</tr>
<tr>
<td></td>
<td>1  5</td>
</tr>
<tr>
<td>;</td>
<td>N  4</td>
</tr>
<tr>
<td>WHILE grouping</td>
<td>J  9</td>
</tr>
<tr>
<td>&lt;=</td>
<td>X  2</td>
</tr>
<tr>
<td>IF = (relational)</td>
<td>0  1</td>
</tr>
<tr>
<td>subscript</td>
<td>2  1</td>
</tr>
<tr>
<td>+</td>
<td>2  2</td>
</tr>
</tbody>
</table>
operators                                      operands

IF  3                                      X  5
<  6                                      Y  5
= (assignment) 4                           Z  6
;  4                                      SMALL 4

n1 = 4                                     n2* = 4
n2 = 4                                     V* = 15.5
TN1 = 17                                   V(PC) = 111
TN2 = 20                                   PC = 995.0

(6B) SMALL = X;
IF Y < SMALL THEN SMALL = Y;
IF Z < SMALL THEN SMALL = Z;

I/O operands: X, Y, Z, SMALL

operators                                      operands

= (assignment) 3                             SMALL 7
;  3                                      X  1
IF  2                                      Y  3
<  4                                      Z  3

n1 = 4                                     n2* = 4
n2 = 4                                     V* = 15.5
TN1 = 12                                   V(PC) = 78
TN2 = 14                                   PC = 392.5

5. (8A) Same as 6B.

(8B) SMALL = AMIN (X, Y, Z);

I/O operands: X, Y, Z, SMALL

operators                                      operands

= (assignment) 1                             SMALL 1
AMIN 1                                      X  1
grouping 1                                  Y  1
;  2                                      Z  1

n1 = 5                                     n2* = 4
n2 = 4                                     V* = 15.5
TN1 = 6                                     V(PC) = 31.7
TN2 = 4                                     PC = 64.9
6. (9A) The intent of this pair of programs is to compare use of the FORTRAN arithmetic IF with other program forms. In order to express this comparison in PL/I without penalizing the poor version 9A, suppose PL/I has a case statement, call it ACASE, which performs like the FORTRAN arithmetic IF, with the exception that transfer of control from the end of each case is handled implicitly.

\[ I = 1; \]
\[ \text{DO WHILE (} I \leq M); \]
\[ \text{ACASE (BP (} I \text{)} + 1.0) \text{ SMALLER, NEGONE, NOTHING;} \]

**SMALLER:**
\[ \text{DO;} \]
\[ \text{IBN1 (} I \text{)} = \text{BLNK;} \]
\[ \text{IBN2 (} I \text{)} = \text{BLNK;} \]
\[ \text{END;} \]

**NEGONE:**
\[ \text{DO;} \]
\[ \text{BP (} I \text{)} = -1.0; \]
\[ \text{IBN1 (} I \text{)} = \text{BLNK;} \]
\[ \text{IBN2 (} I \text{)} = \text{BLNK;} \]
\[ \text{END;} \]

**NOTHING:**
\[ \text{END;} \]

I/O operands: BP, M, IBN1, IBN2

In performing the counting to obtain TN1 and TN2, the labels following the operator ACASE are not counted. This clearly yields software science basic measures which are less than those in the original example.

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (assignment)</td>
<td>I 10</td>
</tr>
<tr>
<td>;</td>
<td>1 2</td>
</tr>
<tr>
<td>WHILE</td>
<td>M 2</td>
</tr>
<tr>
<td>grouping</td>
<td>BP 3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>IBN1 2</td>
</tr>
<tr>
<td>ACASE</td>
<td>BLNK 4</td>
</tr>
<tr>
<td>subscript</td>
<td>IBN2 2</td>
</tr>
<tr>
<td>+</td>
<td>-1 1</td>
</tr>
</tbody>
</table>

n1 = 8
n2 = 8
TN1 = 35
TN2 = 26

n2* = 4
V* = 15.5
V(PC) = 244
PC = 3838.6
(9B)  

\[ I = 1; \]
\[ \text{DO WHILE (I} \leq \text{M);} \]
\[ \text{IF BP (I)} \leq -1.0 \text{ THEN} \]
\[ \text{DO;} \]
\[ \text{BP (I)} = -1.0; \]
\[ \text{IBN1 (I)} = \text{BLNK}; \]
\[ \text{IBN2 (I)} = \text{BLNK}; \]
\[ \text{END;} \]
\[ \text{END;} \]

I/O operands: M, BP, IBN1, IBN2

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (assignment)</td>
<td>I 8</td>
</tr>
<tr>
<td>;</td>
<td>1 1</td>
</tr>
<tr>
<td>WHILE</td>
<td>M 2</td>
</tr>
<tr>
<td>grouping</td>
<td>BP 3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>-1 3</td>
</tr>
<tr>
<td>IF</td>
<td>IBN1 1</td>
</tr>
<tr>
<td>subscript</td>
<td>BLNK 2</td>
</tr>
<tr>
<td></td>
<td>IBN2 1</td>
</tr>
</tbody>
</table>

\[ n1 = 7 \]
\[ n2 = 8 \]
\[ n2* = 4 \]
\[ V* = 15.5 \]
\[ TN1 = 25 \]
\[ V(PC) = 179.7 \]
\[ TN2 = 21 \]
\[ PC = 2085.9 \]

7. (10A) TRAPZ: PROCEDURE OPTIONS (MAIN);

\[ \text{DCL MSSG1 CHAR (20);} \]
\[ \text{MSSG1 = 'AREA UNDER THE CURVE'} ; \]
\[ \text{DCL MSSG2 CHAR (23);} \]
\[ \text{MSSG2 = 'BY THE TRAPEZOIDAL RULE'} ; \]
\[ \text{DCL MSSG3 CHAR (16);} \]
\[ \text{MSSG3 = 'FOR DELTA X = 1/'} ; \]
\[ \text{DCL I FIXED DEC (2);} \]
\[ \text{DCL J FIXED DEC (2);} \]
\[ \text{DCL L FIXED DEC (7, 6);} \]
\[ \text{DCL M FIXED DEC (7, 6);} \]
\[ \text{DCL N FIXED DEC (2);} \]
\[ \text{DCL AREA1 FIXED DEC (8, 6);} \]
\[ \text{DCL AREA FIXED DEC (8, 6);} \]
\[ \text{DCL LMTS FIXED DEC (5, 4);} \]
\[ \text{PUT SKIP (MSSG1) (X (9), A (20));} \]
\[ \text{PUT SKIP EDIT (MSSG2) (X (7), A (23));} \]
\[ \text{PUT SKIP EDIT (' ') (A (1));} \]
\[ \text{AREA = 0;} \]
\[ \text{K = 4;} \]
\[ \text{DO WHILE (K} \leq 10); \]
\[ \text{M = 1/K;} \]
\[ \text{N = K - 1;} \]
\[ \text{LMTS = .5 * M;} \]
I = 1;
J = 1;
DO WHILE (J <= N);
L = (1/K) ** 2;
AREA1 = .5 * M * (2 * L);
AREA = AREA + AREA1;
IF I = N THEN CALL OUT;
ELSE I = I + 1;
J = J + 1;
END;
K = K + 1;
END;
OUT: PROCEDURE;
AREA = AREA + LMTS;
PUT SKIP EDIT (MSSG3, K, AREA) (X (2), A (16),
F (2), X (6), F (9, 6));
AREA = 0;
RETURN;
END;
END;

I/O operands : K, AREA

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (assignment)</td>
<td>MSG1</td>
</tr>
<tr>
<td>;</td>
<td>'AREA...CURVE'</td>
</tr>
<tr>
<td>PUT</td>
<td>MSG2</td>
</tr>
<tr>
<td>WHILE</td>
<td>MSG3</td>
</tr>
<tr>
<td>grouping</td>
<td>'FOR...=1/'</td>
</tr>
<tr>
<td>&lt;= 4</td>
<td>AREA</td>
</tr>
<tr>
<td>/</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>K</td>
</tr>
<tr>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>**</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>10</td>
</tr>
<tr>
<td>IF</td>
<td>M</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
</tr>
<tr>
<td>CALL OUT</td>
<td>N</td>
</tr>
<tr>
<td>RETURN</td>
<td>LMTS</td>
</tr>
</tbody>
</table>

| n1 = 15       | n2* = 2  |
| n2 = 21       | V* = 8.0 |
| TN1 = 75      | V(PC) = 713.5 |
| TN2 = 63      | PC = 63,637 |
(10B) TRAPZ: PROCEDURE OPTIONS (MAIN);
DCL (J, K) FIXED DEC (2),
   AREA FIXED DEC (8, 6);
PUT SKIP EDIT ('AREA UNDER THE CURVE',
   'BY THE TRAPEZOIDAL RULE')
   (X (9), A, SKIP, X (7), A);
PUT SKIP;
K = 4;
DO WHILE (K <= 10);
   AREA = 0.5/K;
   J = 1;
   DO WHILE (J <= K - 1);
      AREA = AREA + ((J/K) ** 2)/K;
      J = J + 1;
   END;
   PUT SKIP EDIT ('FOR DELTA X = 1/',  K, AREA)
      (X (2), A, F (2), X (6), F (9, 6));
   K = K + 1;
END;
END TRAPZ;

I/O operands: K, AREA

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT 3 K 11</td>
<td></td>
</tr>
<tr>
<td>; 12 4 1</td>
<td></td>
</tr>
<tr>
<td>= (assignment) 6 10 2</td>
<td></td>
</tr>
<tr>
<td>WHILE 2 AREA 4</td>
<td></td>
</tr>
<tr>
<td>grouping 6 .5 1</td>
<td></td>
</tr>
<tr>
<td>&lt;= 4 J 6</td>
<td></td>
</tr>
<tr>
<td>/ 3 1 6</td>
<td></td>
</tr>
<tr>
<td>- 2 2 1</td>
<td></td>
</tr>
<tr>
<td>+ 3</td>
<td></td>
</tr>
<tr>
<td>** 1</td>
<td></td>
</tr>
</tbody>
</table>

n1 = 10 n2* = 2
n2 = 8 V* = 8.0
TN1 = 42 V(PC) 308.6
TN2 = 32 PC = 11,904.5

8. (11A) The following code preserves most of the undesirable characteristics of the original code segment with the exception of the unstructured GO TO RETURN statement.

M = 1;
DO WHILE (M <= N);
   K = N - 1;
   J = 1;
   DO WHILE (J <= K);
      IF ARAY (J) ARAY (J + 1) >= 0 THEN;
      ELSE DO,
SAVE = ARAY (J);
ARAY (J) = ARAY (J + 1);
ARAY (J + 1) = SAVE;
END;
J = J + 1;
END;
M = M + 1;
END;

I/O operands: ARAY, N

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (assignment) 8</td>
<td>M 5</td>
</tr>
<tr>
<td>;</td>
<td>1 9</td>
</tr>
<tr>
<td>WHILE</td>
<td>N 3</td>
</tr>
<tr>
<td>grouping</td>
<td>K 2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>J 13</td>
</tr>
<tr>
<td>-</td>
<td>ARAY 8</td>
</tr>
<tr>
<td>IF</td>
<td>0 2</td>
</tr>
<tr>
<td>subscript</td>
<td>SAVE 2</td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
</tr>
</tbody>
</table>

n1 = 10
n2 = 8
TN1 = 49
TN2 = 44

(11B) M = 1;
DO WHILE (M <= N - 1);
   J = 1;
   DO WHILE (J <= N - 1);
      IF ARAY (J) < ARAY (J + 1) THEN
         DO;
            SAVE = ARAY (J);
            ARAY (J) = ARAY (J + 1);
            ARAY (J + 1) = SAVE;
         END;
      END;
   END;
END;

I/O operands: ARAY, N

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (assignment) 5</td>
<td>M 3</td>
</tr>
<tr>
<td>;</td>
<td>1 10</td>
</tr>
<tr>
<td>WHILE</td>
<td>N 4</td>
</tr>
<tr>
<td>grouping</td>
<td>J 11</td>
</tr>
<tr>
<td>&lt;=</td>
<td>ARAY 8</td>
</tr>
<tr>
<td>-</td>
<td>SAVE 2</td>
</tr>
<tr>
<td>IF</td>
<td></td>
</tr>
<tr>
<td>subscript</td>
<td></td>
</tr>
</tbody>
</table>

(continued)
operators

\[<  \quad 2\]
\[+  \quad 4\]

n1 = 10
n2 = 6
TN1 = 46
TN2 = 38

n2* = 2
V* = 8.0
V(PC) = 336.0
PC = 14,112

9. (14A) IF QTY > 10 THEN
    IF QTY > 200
    THEN
        IF QTY >= 500
            THEN BILL_A = BILL_A + 1.00;
        ELSE BILL_A = BILL_A + .50;
    ELSE
    ELSE BILL_A = .00;

I/O operands: QTY, BILL_A

operators

\[IF  \quad 3\]
\[>  \quad 4\]
\[>=  \quad 2\]
\[= \text{(assignment)} 3\]
\[+  \quad 2\]
\[;  \quad 4\]

operands

QTY  6
10   2
200  2
500  2
BILL_A 5
1   2
.5   1
0   1

n1 = 6
n2 = 8
TN1 = 18
TN2 = 21

n2* = 2
V* = 8.0
V(PC) = 148.5
PC = 2756.3

(14B) IF QTY >= 500 THEN BILL_A = BILL_A + 1.00;
ELSE
    IF QTY > 200 THEN BILL_A = BILL_A + .50;
ELSE
    IF QTY <= 10 THEN BILL_A = 0.0;

I/O operands: QTY, BILL_A
operators | operands
---|---
IF | 3 | QTY | 6
>= | 2 | 500 | 2
= (assignment) | 3 | BILL_A | 5
+ | 2 | 1 | 1
; | 3 | 200 | 2
> | 2 | .5 | 1
< | 2 | 10 | 2
0 | 1

n1 = 7
n2 = 8
TN1 = 17
TN2 = 20

n2* = 2
V* = 8.0
V(PC) = 144.6
PC = 2612.8

10. (15A) IF X >= Y
THEN
IF Y >= Z THEN SMALL = Z;
ELSE SMALL = Y;
ELSE
IF X >= Z THEN SMALL = Z;
ELSE SMALL = X;

I/O operands: X, Y, Z, SMALL

operators | operands
---|---
IF | 3 | X | 5
>= | 6 | Y | 5
= (assignment) | 4 | Z | 6
; | 4 | SMALL | 4

n1 = 4
n2 = 4
TN1 = 17
TN2 = 20

n2* = 4
V* = 15.5
V(PC) = 111
PC = 794.9

(15B) Same as 6B.

11. (18A) The poor version of Gordon's example program is unstructured and hence PC cannot be determined precisely. However, it is clear that a measure which, in addition to the counts of operator and operand occurrences, accounts for control flow complexity must yield a value greater than that which uses the counts alone. This is a general observation, and a comparison of Gordon's E measure with PC for the programs in Appendix A does not contradict this result. Therefore, it must be the case that an appropriate PC-like value for Gordon's example program 18A must be greater than 94,677, which is E' for that program.
(18B) DATES: PROCEDURE OPTIONS (MAIN);
   DCL MONSIZE (0:1, 1:12) INITIAL (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
   DO WHILE ('1'B);
      GET LIST (IYEAR, IDATE) COPY;
      IF MOD (IYEAR, 400) = 0 |
         (MOD (IYEAR, 100) | MOD (IYEAR, 4) = 0) THEN LEAP = 1;
      ELSE LEAP = 0;
      IF (IYEAR < 1753 | IYEAR > 3999 | IDATE = 0 | IDATE > 365 + LEAP) THEN PUT SKIP LIST ('BADYEAR, DATE - ', IYEAR, IDATE);
      ELSE DO;
         NDAYS = 0;
         MONTH = 1;
         DO WHILE (MONTH <= 12 & IDATE > NDAYS + MONSIZE (LEAP, MONTH));
            NDAYS = NDAYS + MONSIZE (LEAP, MONTH);
            MONTH = MONTH + 1;
         END;
         PUT SKIP LIST (MONTH, IDATE-NDAYS, IYEAR);
      END;
   END;
END DATES;

I/O operands: IYEAR, IDATE, MONTH, (IDATE-NDAYS)

<table>
<thead>
<tr>
<th>operators</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE</td>
<td>2</td>
</tr>
<tr>
<td>grouping</td>
<td>13</td>
</tr>
<tr>
<td>;</td>
<td>15</td>
</tr>
<tr>
<td>GET</td>
<td>1</td>
</tr>
<tr>
<td>IF</td>
<td>2</td>
</tr>
<tr>
<td>MOD</td>
<td>6</td>
</tr>
<tr>
<td>= (relational)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>= (assignment)</td>
</tr>
<tr>
<td>&lt;</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>6</td>
</tr>
<tr>
<td>&lt;=</td>
<td>4</td>
</tr>
<tr>
<td>++</td>
<td>6</td>
</tr>
<tr>
<td>PUT</td>
<td>2</td>
</tr>
<tr>
<td>&amp;</td>
<td>2</td>
</tr>
<tr>
<td>subscript</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>
\begin{align*}
n_1 &= 20 \\
n_2 &= 15 \\
\text{TN}_1 &= 98 \\
\text{TN}_2 &= 69 \\
n_2^* &= 4 \\
v^* &= 15.5 \\
V(\text{PC}) &= 856.6 \\
\text{PC} &= 51.609.1
\end{align*}