NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

CYCLOMATIC COMPLEXITY AS A UTILITY
FOR PREDICTING SOFTWARE FAULTS

by

Edwin Arthur Shuman IV

March, 1990

Thesis Advisor: Timothy J. Shimeall

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) 55	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (*If applicable*)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (*City, State, and ZIP Code*)	10 SOURCE OF FUNDING NUMBERS Program Element No. / Project No. / Task No. / Work Unit Accession Number

11 TITLE (*Include Security Classification*)
CYCLOMATIC COMPLEXITY AS A UTILITY FOR PREDICTING SOFTWARE FAULTS

12 PERSONAL AUTHOR(S) Shuman, Edwin, A. IV

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (*year, month, day*)	15 PAGE COUNT
Master's Thesis	From To	March 1990	96

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Cyclomatic Complexity

19 ABSTRACT (*continue on reverse if necessary and identify by block number*)

The cyclomatic complexity provides a means of quantifying intra-modular software complexity, and its utility has been suggested in the software development and testing process. In this study, an empirical analysis was undertaken to examine the relationship between the cyclomatic complexity and the incidence of faults in a series of eight relatively large (from 1200 to 2400 LOC), complex programs. Each of these programs was developed from a single program specification and subsequently subjected to rigorous unit level testing. A comparison was also made between the relationship of cyclomatic complexity to faults and Lines of Code (LOC) to faults.

The results of this study support a relationship between the cyclomatic complexity and the incidence of faults. Further, a relationship between LOC and faults is demonstrated. It could not be shown that there exists a stronger relationship between cyclomatic complexity and faults than LOC and faults.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ [illegible] ☐ [illegible] ☐	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Timothy J. Shimeall	22b TELEPHONE (*Include Area code*) (408) 646-2509

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted. SECURITY CLASSIFICATION OF THIS PAGE

All other editions are obsolete.

Approved for public release; distribution is unlimited.

Cyclomatic Complexity as a Utility for
Predicting Software Faults

by

Edwin Arthur Shuman IV
Lieutenant, United States Navy
B.A., University of Virginia

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
March 1990

Author: _____
Edwin Arthur Shuman IV

Approved by: _____
Timothy J. Shimeall, Thesis Advisor

_____
William J. Haga, Second Reader

_____
David R. Whipple, Chairman
Department of Administrative Sciences

ABSTRACT

The cyclomatic complexity metric provides a means of quantifying intra-modular software complexity, and its utility has been suggested in the software development and testing process. In this study, an empirical analysis was undertaken to examine the relationship between the cyclomatic complexity and the incidence of faults in a series of eight relatively large (from 1200 to 2400 LOC) complex programs. Each of these programs was developed from a single program specification and subsequently subjected to rigorous unit level testing. A comparison was also made between the relationship of cyclomatic complexity to faults and the relationship of Lines of Code (LOC) to faults.

The results of this study support a relationship between the cyclomatic complexity and the incidence of faults. Further, a relationship between LOC and faults is demonstrated. It could not be shown that there exists a stronger relationship between cyclomatic complexity and faults than LOC and faults.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Prof. Timothy J. Shimeall for his efforts in providing the programs used as the basis for this study, and for his guidance and instruction that made possible the successful completion of this work.

I would like to thank Prof. William J. Haga for his contribution, in providing a thorough review of this thesis.

I would like to thank LCDR John Yurchak, USN for his assistance in resolving some of the programming aspects of this project.

I would like to thank Nancy Leveson for her efforts in providing the programs used in this study.

I. INTRODUCTION

A. IMPORTANCE OF SOFTWARE TESTING

The question of software accuracy and reliability has become an important issue facing software developers, especially in the last decade. There has been an explosion in the demand for new software systems. Software is being applied to all facets of life, from relatively commonplace and unsophisticated transaction systems to unforgiving and time critical military and space applications.

While hardware costs have gone down in the past decade, software costs are on the increase, mainly because software is being applied to increasingly complex and ever larger systems. Brooks amusingly likens the unforgiving environment of software development to the precise and unforgiving incantations of the mythical medieval sorcerer. As with the incantations, the program must be constructed with absolute precision. [Ref. 1] Software can be seen as an almost magical solution to a multitude of problems, but software development, with all the benefits of structured design methodologies and programming tools, is still an activity that is highly human intensive and subject to all the vicissitudes and imperfections that go along with being human.

The consequences of even small software mistakes can be quite large. For example, there is the commonly told story of the spacecraft "Pioneer" that was unable to carry out its multimillion-dollar mission because of a single misplaced character. As costly systems become software dependent, both in terms of monetary value and the potential cost to human life, a system software must commensurately become more reliable and cost efficient. Software, ideally, should be reliable both in short term performance and in its ability to accommodate future requirements (maintenance). McCabe states:

> "That the issues of testability and maintainability are important is borne out by the fact that we often spend half of the development time in testing." [Ref. 2]

In the same vein Boehm states:

> "In 1985, software costs totaled roughly $11 billion in the US Department of Defense, $70 billion in the US overall, and $140 billion worldwide. If present software cost growth rates of approximately 12 percent per year continue, the 1995 figures will be $36 billion for the DoD, $225 billion for the US, and $450 billion worldwide. Thus even a 20 percent improvement in software productivity would be worth $45 billion in 1995 for the US and $90 billion worldwide." [Ref. 3]

Thus an improvement in testing that results in a 20 percent gain in efficiency would be worth somewhere between 28 and 90 billion dollars in terms of worldwide savings annually. The ability to identify regions of code where software testing

effort should be concentrated could save developers both time and money, or result in more reliable systems for the same money.

Software testing, like other phases of software development such as design, coding and debugging, is an activity that is extremely demanding of human resources. It is not the sort of activity that can be greatly improved upon through the introduction of better or faster hardware, for example. Furthermore, complete testing is not theoretically possible. Three reasons are given for this inability to completely test a program in Beizer's "Software System Testing and Quality Assurance":

> "-We can never be sure that the specifications are correct.
> -No verification system can verify every correct program.
> -We can never be certain that a verification system is correct." [Ref. 4]

Given that testing is an activity that is human resource intensive and that theoretically a program can never be completely tested, then it is reasonable to search for some tool to guide testing activities to minimize commitment of human resources and money. McCabe suggests the cyclomatic complexity as just such a tool. [Ref. 2]

B. APPROACH

1. Definition of Cyclomatic Complexity

The cyclomatic complexity is a mathematical technique that provides a quantitative basis for modularization and for developing a testing strategy, by evaluating source code in terms of logical decision points. The cyclomatic complexity is a measure of the number of paths in a program. One problem with measuring the possible number of paths is that where a backwards branch exists, there is the possibility of an infinite number of paths. Using the total number of paths is not a realistic approach. Therefore cyclomatic complexity is defined as the number of basic paths through a program. The cyclomatic complexity (v(G)) of a program is derived by associating a directed graph with it. In figure 1, a directed graph, each node (a - e) represents a block of code where the flow is sequential, and the arcs represent branches in logical program structure. [Ref. 2]

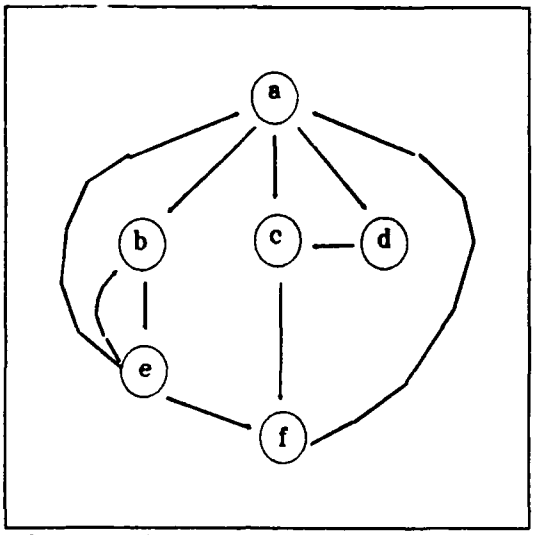

Figure 1 Control Graph

The McCabe complexity measure is based on the application of a graph-theoretic complexity measure to a program's structure.

"Definition 1: The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

$$v(G) = e - n + p.$$

Theorem 1: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits." [Ref. 2]

Referring to figure one, an example of a linearly independent circuit in G is: a-c-f-a, where the starting point is a and the ending point is a. McCabe derives a simplified form of the cyclomatic complexity equation. In this simplified equation, the cyclomatic complexity (v(G)) equals the number of predicates (pi) plus one, where each predicate is a logical

decision point such as "if-then" (refer to chapter 3 for a complete list of Pascal predicates to be evaluated):

$$v(G) = \pi + 1.$$

This simplified form is easily automated in a tool for obtaining the cyclomatic complexity of a block of code. The cyclomatic complexity measure can be determined for each of the functions and procedures of a given program. The testing effort on the procedures and functions could, therefore, be concentrated on the basis of higher complexity (v(G)) values. The assumption would be that there is potentially a higher number of errors in these more complex procedures and functions. McCabe suggests a v(G) value of ten, as an indicator of where testing effort should be concentrated. [Ref. 2]

2. Cyclomatic Complexity Measure and Lines of Code

McCabe goes on to suggest that the measure of cyclomatic complexity is of greater utility as an indicator of potentially troublesome regions of code than the simple measure of lines of code (LOC). Shortness does not always imply simplicity. A short module, consisting of a series of twenty five if-then's (i.e., a high cyclomatic complexity) has a large number of possible logical paths.

The cyclomatic measure, representing the inherent complexity of a module, would be a more reliable predictor of potentially faulty regions of code than the simple lines of code (LOC) metric. [Ref. 2]

3. Use of An Experiment to Evaluate the Cyclomatic Complexity

In light of the theoretical work developed by McCabe, the question remains as to whether there is a correlation between the complexity metric and what is actually born out through empirical analysis. There is also the question of how the cyclomatic complexity metric compares with other metrics.

Few experiments have examined the incidence of errors with respect to the cyclomatic complexity measure. In a study by Walsh, the occurrence of errors was measured against cyclomatic complexity in eight functionally related modules (essentially one large program). These modules were comprised of 276 procedures or programs making up a software control loop and display processing for the Aegis Naval Weapons System. The error data were derived through the compilation of trouble reports. In the study, Walsh endeavored to distinguish between "software errors" and "design errors" by interviewing programmers responsible for the code. The study divided the procedures and programs into two groups, one group where cyclomatic complexity was less than ten and the other

group where it was greater, in order to compare the incidence of errors in the two groups. It was found that there was a 21% higher incidence of errors in the group with a higher cyclomatic complexity. [Ref. 5]

In a study by Schneidewind, four programs were examined for the correlation between the incidence of errors and the cyclomatic complexity. The four programs were programmed by the same programmer in Algol for execution on the IBM 360/67. Program one was 141 source statements. Program two was 712 source statements. Program three was 70 source statements and program four was 1084 source statements. It can be seen that three of the programs were fewer than 1000 source statements; one was just barely over that number. Two of the programs were in fact quite small (one and three). The error data for these experiments were obtained through programmer-generated error reports. The experiment concluded that program structure (v(G)) would have a significant effect on the number of errors. In this study it was further stated that the relationship between errors and structure was not expressible as a mathematical function but serves to partition structures into high or low occurrence depending on whether the cyclomatic complexity is high or low. [Ref. 6]

In a study by Basili, a database of ground support software for satellites was studied to investigate the

relationship between program cyclomatic complexity and errors. In this study Basili examined systems ranging from 51,000 to 112,000 lines of Fortran source code. Each system was made up of from 200 to 600 modules (a total of 1794 modules were analyzed in the entire experiment). Error data were collected from change report forms. It was the conclusion of this study that there was a poor correlation of cyclomatic complexity with errors in these large programs. Basili cites a Spearman rank order correlation for cyclomatic complexity with faults of .196. The low correlations were attributed to the fact that 340 of the 652 modules analyzed with regard to errors had zero reported errors. [Ref. 7]

In a study by Ward, two large scale programs were examined for correlation of errors to cyclomatic complexity at Hewlett Packard's Walthan Division. Projects A and B were 125,000 and 77,000 lines of non-commented source code respectively. The two programs were independent of each other, had short development times and had a low post-release defect density. The error database for the experiment was compiled from prerelease development data that had been maintained on the programs. Ward concluded, that a .8 statistical correlation was found between complexity and defect density. [Ref. 8]

In a study by Butler, four small programs from operational ECM Software were studied. Program one was 118 lines of code. Program two was 89 lines of code.

Program three was 173 lines of code and program four was 135 lines of code. For each of the four programs a qualitative assessment of the programs was made in terms of maintainability, readability, comprehensibility and "bug detection". The data for this qualitative assessment were made over a period of two years on the basis of customer comments. It was the conclusion of this study that the cyclomatic complexity measure provides the "best" method to analyze and limit complexity of a software program. No direct correlation was found between the incidence of errors and cyclomatic complexity. A broad qualitative relationship was indicated. [Ref. 9]

In a study by Meals, tools to automate complexity measures were coded in COBOL and applied against three, separate, large commercial COBOL programs. These programs were examined to determine if there was a correlation between cyclomatic complexity and the known error history of the software. In the course of the development of program one, it grew from 4956 lines to 5948. Program two grew from 4239 lines to 5001 lines. Program three grew from 9416 lines to 9425 lines. Two of the three programs showed correlations between the incidence of errors and the cyclomatic complexity (however no coefficient was given). The conclusion was that cyclomatic complexity was useful as an indicator of error-prone sections of code. [Ref. 10]

Henry, Kafura and Harris undertook a study of a single, large system (the UNIX operating system) in which cyclomatic complexity was compared to the occurance of errors. A list of errors was obtained from the UNIX User's Group. A strong correlation (.95) was found to exist between procedures containing errors and cyclomatic complexity. [Ref. 11]

Gollhofer, Shimeall and Leveson examined cyclomatic complexity with respect to errors in a group of 27 independent implementations of a multi-version software experiment. These programs ranged in size from 400 to 800 LOC and had an average execution time of three minutes. Faults were derived through previous testing studies. This study concluded that there was a strong correlation between higher v(G) (v(G) > 10) and the incidence of errors. Nine percent of the modules had a v(G) greater than ten; these same modules had 47% of the errors. [Ref. 12]

The previous experiments that attempted to correlate the cyclomatic complexity measure with the incidence of errors fall into two groups: empirical studies on a series of relatively small (less than 1000 lines) programs using simple debugging efforts, and studies on a single

large program, using problem reports generated by in-field use. In general, the empirical work done to date suggests that there is a correlation between the incidence of errors in coding and the cyclomatic complexity measure. However, there has been no experiment that examined the correlation between the cyclomatic measure and the incidence of programming errors in a series of independently developed versions of a relatively large, complex program (larger than 1500 lines), each program having been subjected to strenuous fault detection efforts using a variety of detection techniques.

C. PROBLEM STATEMENT

Larger programs tend to manifest a greater incidence of errors by virtue of the increased number of inter-dependencies that tend to be created. [Ref. 13] Therefore, the questions which have been approached in this endeavor are:

- What is the predictive relationship between the cyclomatic measure of program complexity and the incidence of errors in a series of larger programs, i.e, does the relationship between the complexity measure and the incidence of errors hold true in larger programs as it does in smaller ones?

This question addresses how well previous studies scale up: large programs are more than a set of concatenated small ones. In this study, the programs will be analyzed on a version-by-version basis, rather than mixing procedures together. Also, complex applications exibit interdependencies and need to be examined as well.

- How do the results for the complexity metric compare with other similar metrics, notably lines of code (LOC)? Do the results of the comparison of cyclomatic complexity to errors and LOC to errors seem to parallel each other, or does cyclomatic complexity show a better correlation with errors than LOC?

It is important to place results in perspective by use of other measures. The common complexity measure is LOC. It would also be interesting to test McCabe's assertion regarding the utility of LOC as a metric. Lastly, it is reasonable to test what is generally known to be a high correlation between cyclomatic complexity and LOC.

D. OVERVIEW OF THE THESIS

The remaining chapters of this thesis are structured as follows: **Chapter II** provides a description of the study undertaken, presentation of data and statistical analysis, and a discussion of the data with an interpretation of results. **Chapter III** provides a summary of results and conclusions.

II. DESCRIPTION OF EXPERIMENT AND RESULTS

A. INTRODUCTION

This chapter presents the methodology used in this experiment, and presents the data and conclusions generated in light of the questions pcsed in chapter one. In review these questions were:

- What is the correlation between the v(G) of a procedure or function and the incidence of errors?
- How does the cyclomatic complexity compare with Lines of Code (LOC) as a predictor of errors?

B. EXPERIMENTAL METHODOLOGY

1. TOOL FOR COMPUTING V(G)

To make the comparison between the incidence of found errors and v(G), the v(G) had to be determined for each procedure and function within each of the eight programs, described in section C of this chapter. In accordance with the formula:

$$v(G) = \pi + 1$$

cyclomatic complexity was calculated on the basis of predicates (pi) + 1. The following Pascal expressions constitute the predicates that were counted:

if-then's
or's
while
for
case (conditions)
repeat

Pascal Predicates

Because of the size of the programs involved it was decided that the best approach to determining the number of predicates would be to develop a program to automate the process on a modular (procedure and function) basis. The basis for development of such a program is lexigraphical analysis. The C programming language was chosen for this application.

A software tool was then designed and coded to comprise two related functional components. The first functional component of the program is the lexical analyzer. The lexical analyzer reads in the input stream (program) one character at a time. On the basis of the Standard Pascal Reserved Word List, the lexical analyzer identifies what in lexigraphical parlance are termed "tokens". One token is returned each time the lexical analyzer is called. The lexical analyzer was constructed such that each output token is composed of

the stream of characters making up the token and the token identity. The second functional component of the program is the parser. The parser calls the lexical analyzer, and through recursive descent the parser analyses the tokens in terms of all the legal constructs in the Standard Pascal language. Lastly on the basis of the recognition of the logical constructs the predicates are counted. The program generates each procedure or function name and the associated v(G) values.

C. EXPERIMENTAL DATA BASE

This experiment used eight Pascal programs (refer to section B for a description) that had already been subjected to rigorous debugging efforts. These programs were written for and were the subject of Shimeall's [Ref. 14] comparison of fault elimination and fault-tolerance techniques and comparison of various testing techniques in terms of fault detection. They were written from a single specification for a combat simulation problem, derived from an industrial specification. Development followed a standard, controlled, software life cycle approach. The development involved 26 upper division computer science students working in pairs. Eight versions were eventually produced that were determined to be adequate for the purposes of the fault detection, by

successfully executing a 15 case minimal acceptance test. Each program had been tested for errors and 255 faults had been identified and corrected, while preserving the originally faulty code by using conditional compilation flags.

Table I describes the eight versions. The Modules column gives the number of Pascal functions or procedures in each version. The size of each version is given in terms of lines of source code and code lines. Code lines is the size of each version without comments and blank lines. Lastly, the number of errors detected in each version is given in the errors column.

TABLE I VERSION SOURCE PROFILE

Version	Modules	Source	Code	Errors
1	72	7503	2414	35
2	56	3452	1540	11
3	41	1480	1201	33
4	57	3663	2003	26
5	28	1634	1544	25
6	72	3065	2206	24
7	75	2734	1976	23
8	57	1896	1331	16

Each version of the combat simulation program was subjected to five fault detection techniques. These fault detection methodologies were: code reading by stepwise abstraction,

static data flow analysis, run-time assertions inserted by the program development participants, multi-version voting, and functional testing with follow-on structural testing. A fault was considered to exist where there was at least one associated identifiable misbehavior (or failure to perform a required function) in the program. A section of code was considered a single fault (or bug) where it's correction eliminated at least one failure. Several faults could possibly contribute to a failure for a particular data set, and several failures could be caused by a single fault. [Ref. 14]

In the course of counting the previously identified faults, if there was more than one correction of code associated with the correction of a particular misbehavior, then only one fault was counted.

In summary, this data set is unique because it provides eight independently developed versions of a single specification for a program. Because all programs were developed from a single specification, the variability of faults due to differences in design specifications can be eliminated. Each version has been thoroughly debugged using the fault detection techniques. The faults in each function and procedure have been identified. Finally, the application is complex, with many input variables and much iteration, and each version is larger than 1200 lines of code (see Table I).

The variability of faults with respect to cyclomatic complexity has not been tested on multiple program versions of this length, and complexity.

D. RESULTS

Each of the programs were analyzed individually (as opposed to grouping all the procedures and functions together from all eight programs) in order to allow individual program variation to be seen, thereby preventing the masking of one or more program variances by the entire group. Also, in order to determine if other metrics (i.e., LOC) predict faults better than cyclomatic complexity, it was necessary to look at multiple program cases, to estimate the population standard deviation.

The non-parametric tests (ANOVA & Means) were chosen for the statistical analysis instead of a parametric test because no assumptions can be made about the population being normally distributed. These tests are relatively insensitive to the violation of normality.

1. Relationship of V(G) to Faults

The first question under investigation concerns the correlation between the incidence of faults and the value of v(G).

a. *Summary of Observed Data*

Figures 2 - 9 are scatterplots depicting the cyclomatic complexity (on the x axis) in relation to the incidence of faults (on the y axis) in the procedures and functions, for each of the eight programs. Each axis is labelled with a number indicating the maximum value for that variable for each program. An asterisk represents one occurrence at the indicated x,y position. A number represents the number of functions or procedures occurring at the indicated x,y position. Lastly, the pound sign (#) is indicative of ten or more occurrences of functions or procedures at the indicated position.

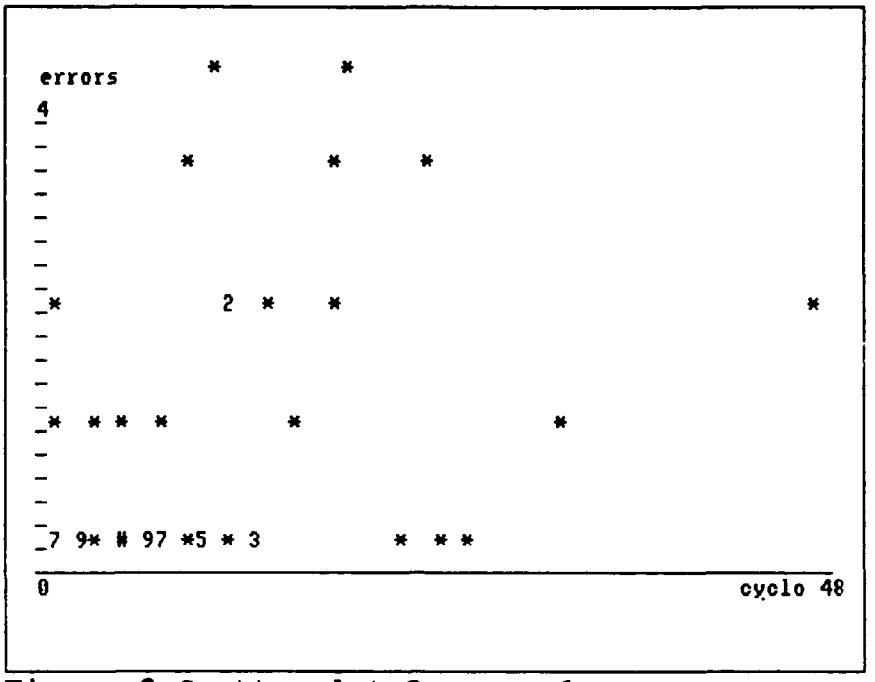

Figure 2 Scatterplot Program 1

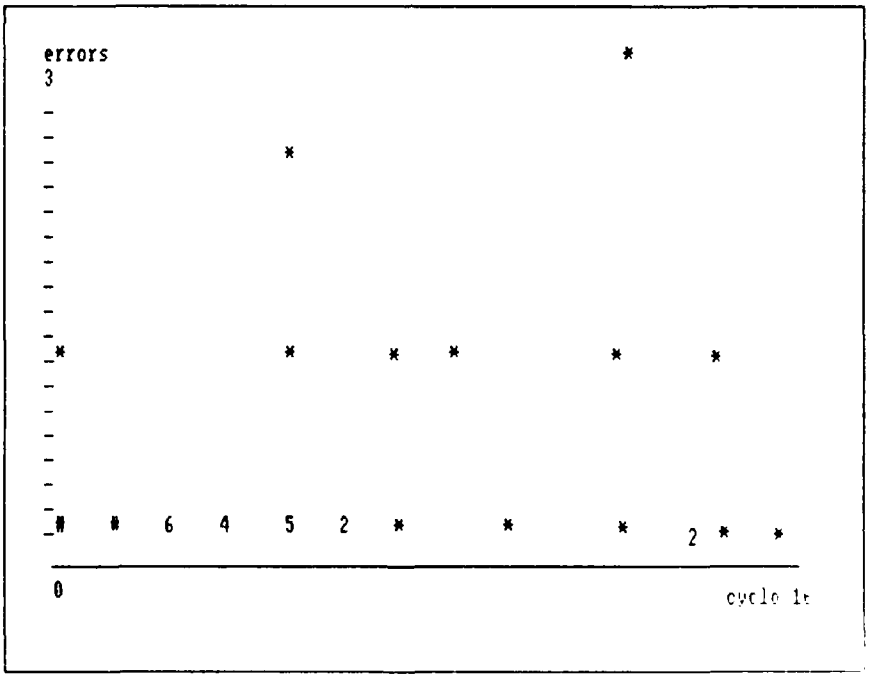

Figure 3 Scatterplot Program 2

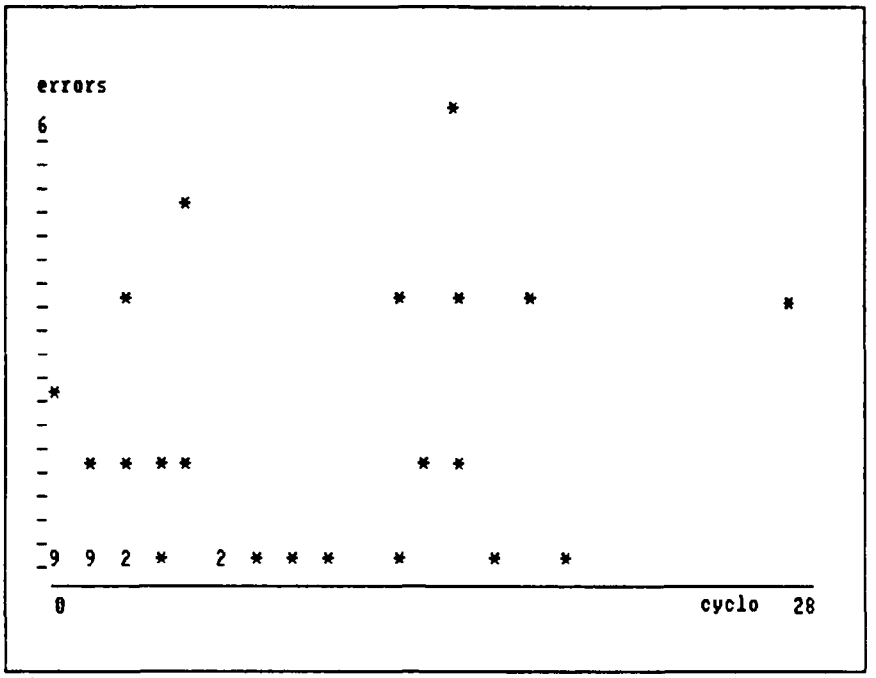

Figure 4 Scatterplot Program 3

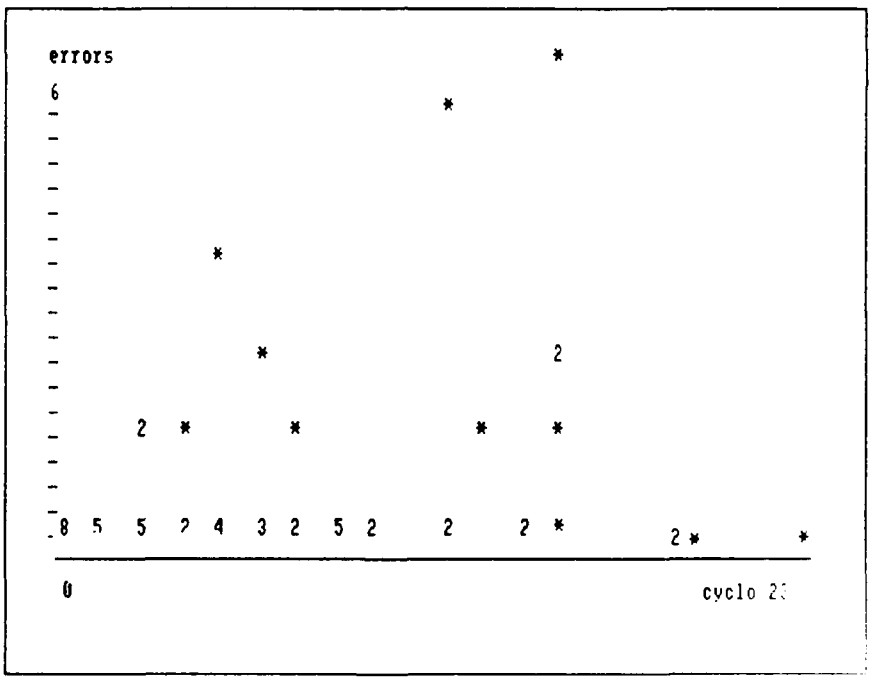

Figure 5 Scatterplot Program 4

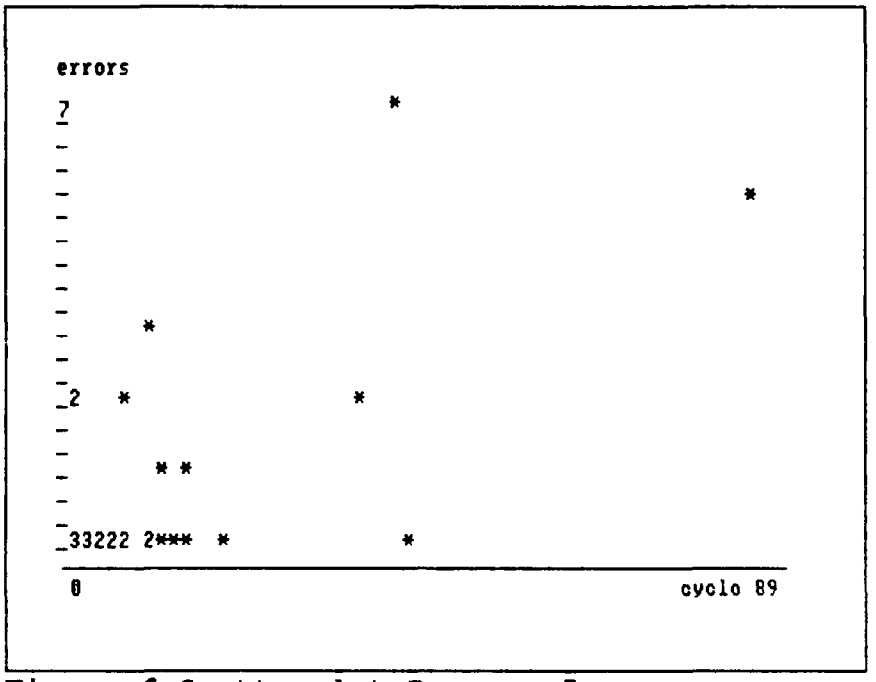

Figure 6 Scatterplot Program 5

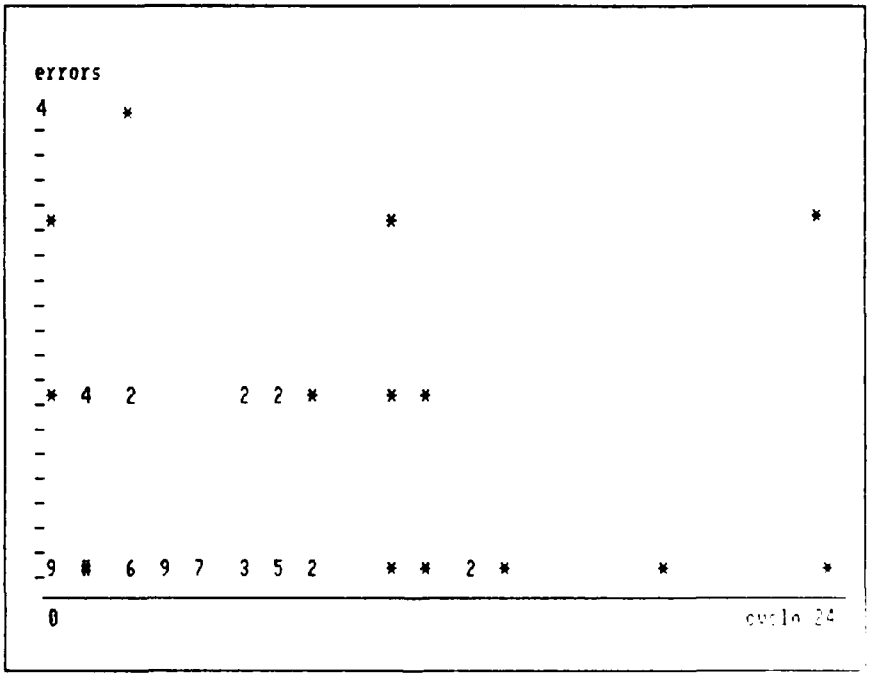

Figure 7 Scatterplot Program 6

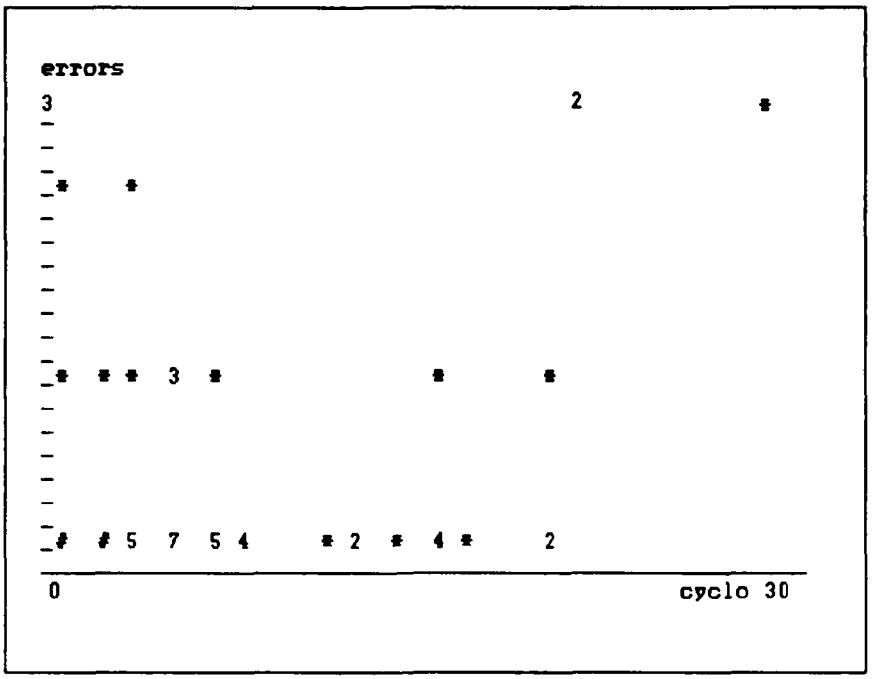

Figure 8 Scatterplot Program 7

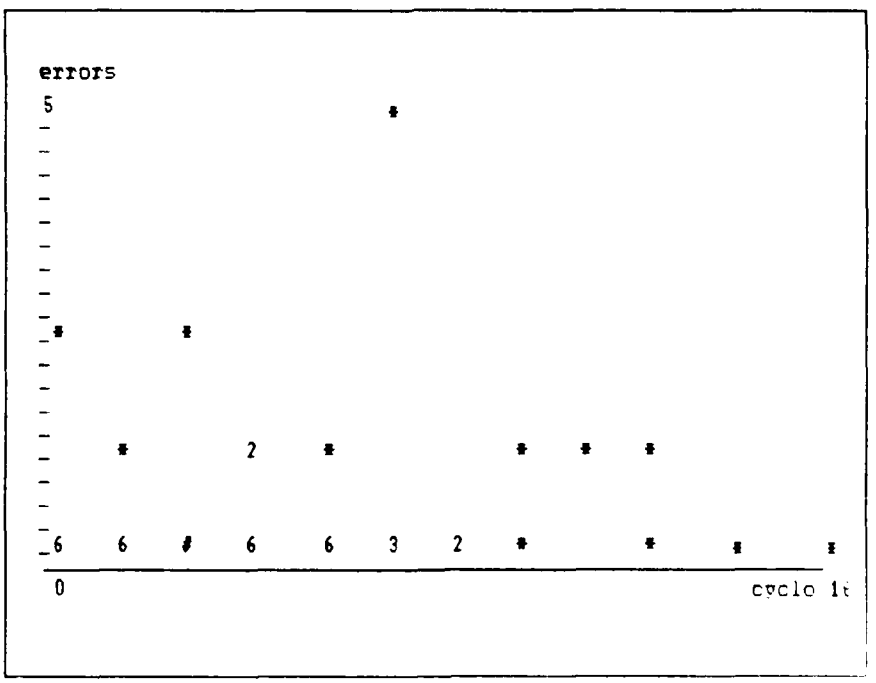

Figure 9 Scatterplot Program 8

There is probably not a linear relationship between cyclomatic complexity and faults, but inspection of the data indicated that the majority of the functions and procedures with faults have a cyclomatic complexity greater than seven. There are, however, a small number of faulty procedures and functions where there is a cyclomatic complexity less than seven.

b. Analysis of Observed Data

To further explore what appeared, through inspection of the data, to be a relationship between higher values of v(G) and faults, two statistical tests were applied to the data: the **ANOVA Test** and the **Means Test**.

(1) ANOVA Tests. Functions and procedures for each program were first ordered according to v(G), from least to greatest, with each associated fault count, and then were divided into thirds:

- v(G) fewer than six, the fault count mean of which is designated :

$$\mu 1$$

- v(G) from six to ten, the fault count mean of which is designated:

$$\mu 2$$

- v(G) greater than ten, the fault count mean of which is designated:

$$\mu 3$$

The null and alternative hypotheses in the one-factor ANOVA model are:

Ho1: $\mu 1 = \mu 2 = \mu 3$

Ha1: at least two of the population means are not equal.

Table II provides a summary of the resultant ANOVA Test probabilities.

TABLE II ANOVA V(G) VS. FAULTS

Version	Probability
1	.00001308
2	.10820000
3	.00288300
4	.02930000
5	.10570000
6	.37630000
7	.00151100
8	.12800000

Ho1 can be rejected on the basis of four programs (one, two, four and seven) at the .05 alpha level. Additionally, if the alpha level of acceptance is broadened to .10 the rejection of Ho1 is supported on the basis of programs three and five). Program eight is just beyond the .10 alpha acceptance level and does not support the rejection of Ho1. The mean value for all eight probabilities is .0940 with a standard deviation of .1260.

This mean value of .0940 does not support the rejection of Ho1 at the .05 alpha level of confidence. If, however the alpha level is broadened to .1 then the mean for the ANOVA probabilities does support the rejection of Ho1. The ANOVA probability of program six is clearly an outlier. It is more than two standard deviations (standard deviation = .1260) from the mean value (.0940) for all eight probabilities generated by the ANOVA tests. The behaviors that make version six an outlier are interesting, and are explored at the end of this chapter. If it is disregarded then the mean value probability for the remaining programs is .05 with a standard deviation of .05. The mean value of this probability supports the rejection of Ho at the .05 level. In summary, (taking into account program six) the ANOVA tests do support the rejection of Ho1, and do support the acceptance of Ha1, the hypothesis that the differences between the numbers of faults in the sub-groupings of procedures and functions is due to more than chance. Because the procedures and functions for each program were divided into three groups on the basis of cyclomatic complexity, a relationship between faults and cyclomatic complexity is indicated.

(2) Means Tests. To further substantiate the results obtained with the ANOVA tests with regard to the relationship between the cyclomatic complexity and the incidence of faults, a means test was similarly performed on

each of the eight programs. The t distribution was used in this test because the fault population standard deviation is unknown. The procedures and functions were ordered according to v(G) as with the ANOVA Tests, and then were divided into two groups, those with a v(G) less than six and those with a v(G) greater than five.

- v(G) less than six, the fault count mean of which is designated :

  $\sigma 1$

- v(G) greater than five, the fault count means of which is designated:

  $\sigma 2$

The null and alternative hypotheses in the Means model are:

Ho2: $\sigma 1 = \sigma 2$

Ha2: $\sigma 1 \neq \sigma 2$

Table III provides a summary of the resultant Means Tests probabilities.

TABLE III MEANS V(G) VS. FAULTS

Version	Probability
1	.0029840
2	.0212000
3	.0251000
4	.0625000
5	.0291000
6	.0825000
7	.0225000
8	.0492000

On the basis of the probabilities derived through the Means Tests, Ho2 can be rejected on the basis of six programs (one, two, three, five, seven and eight), at the .05 alpha level. If the alpha acceptance level is extended to .1, programs four and six support the rejection of Ho2. The mean value for all eight probabilities is .0365 with a standard deviation of .0264, and it supports the rejection of Ho2 at the .05 level.

In summary, the Means Tests do reject Ho2 and support the acceptance of Ha2, the hypothesis that the differences between the mean values of the subgroupings of procedures and functions is due to more than chance. In

other words, a relationship between higher values of cyclomatic complexity and the increased incidence of faults is suggested by both the ANOVA Tests and the Means Tests.

It is important, however, to place this support in context. In real testing efforts, tests are carefully planned, and part of this planning involves the use of metrics. Thus, whether cyclomatic complexity predicts faults in isolation is less useful than whether it predicts better than other commonly used metrics. This issue is explored in the next section.

2. Relationship of Cyclomatic Complexity to Lines of Code

The second question under investigation concerns the comparison of cyclomatic complexity with Lines of Code (LOC) as a predictor of faults. In order to make this comparison, the incidence of faults in the procedures and functions for each program was examined in comparison to the respective LOC for each procedure and function.

a. *Summary of Observed Data*

Figures 9 - 16 are scatterplots depicting the Lines of Code, on the x axis, as a function of the incidence of faults, on the y axis, of the procedures and functions, for each of the eight programs. Each axis is labelled with a number indicating the maximum value for that variable for each program. An asterisk represents one occurrence at the indicated x,y position. A number represents the number of functions of procedures occurring at the indicated x,y position. Lastly, the pound sign (#) is indicative of ten or more occurrences of functions and procedures at the indicated x,y position.

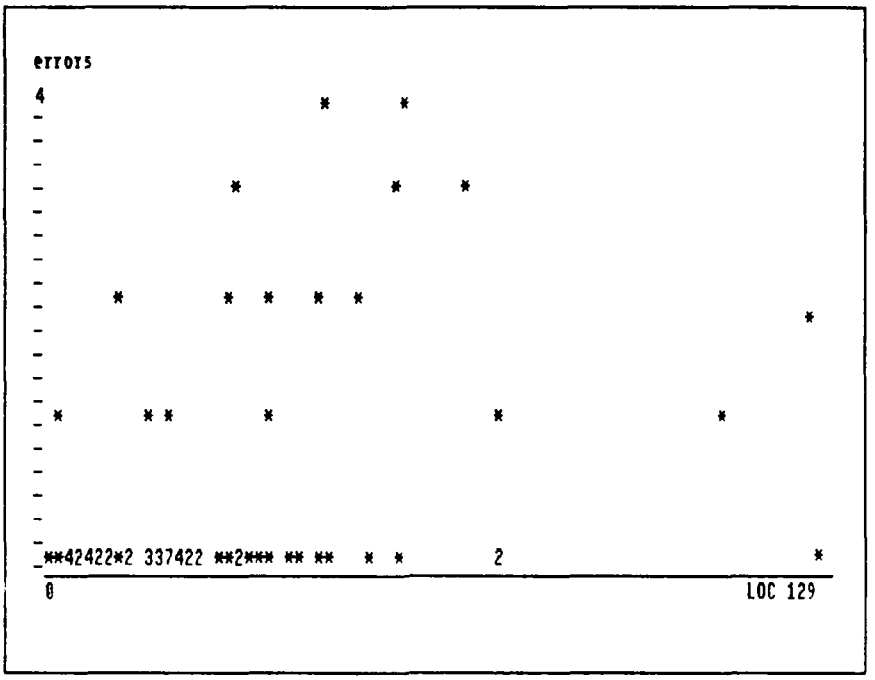

Figure 10 Scatterplot Program 1

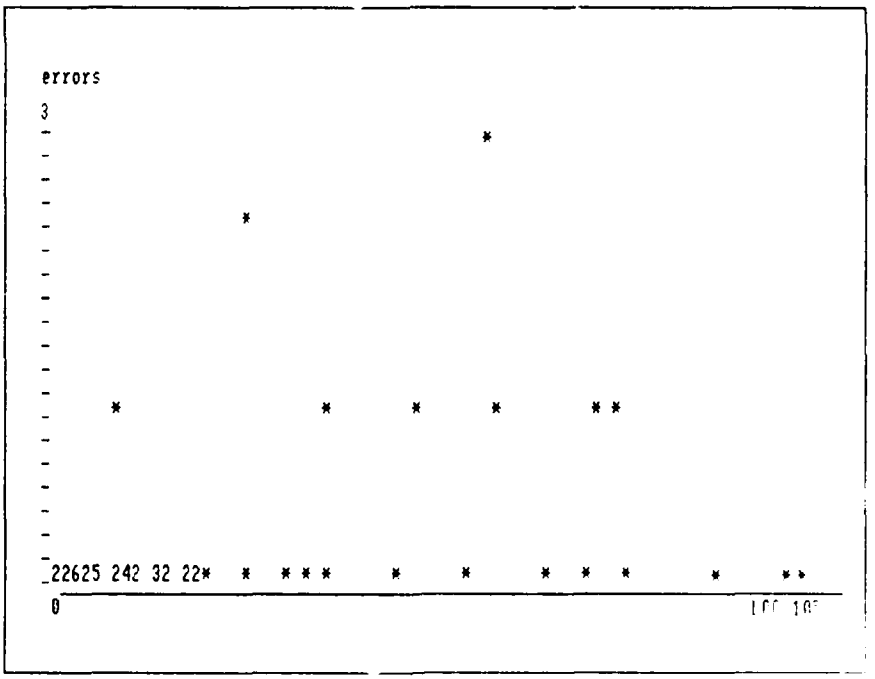

Figure 11 Scatterplot Program 1

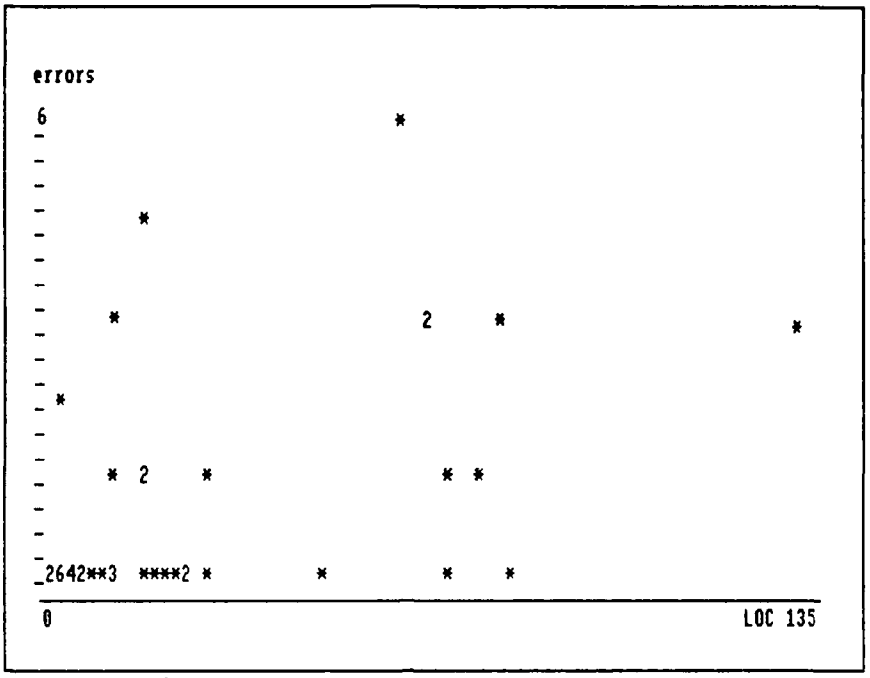

Figure 12 Scatterplot Program 1

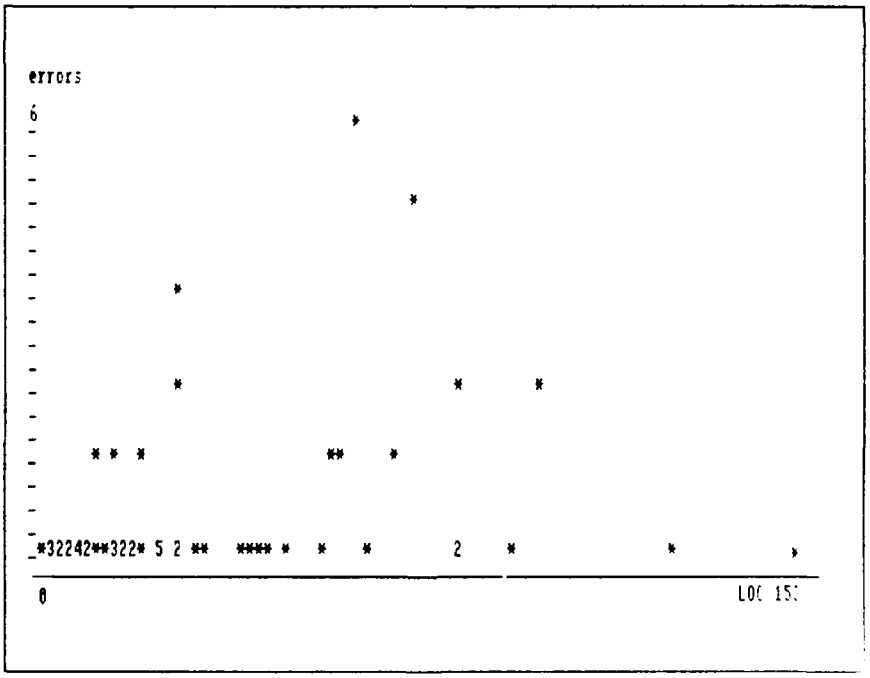

Figure 13 Scatterplot Program 1

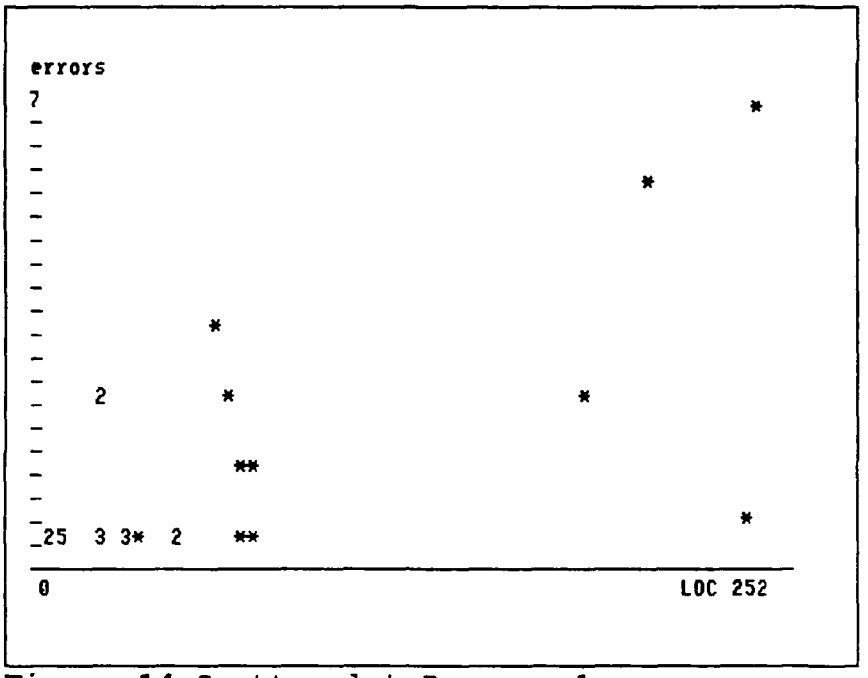

Figure 14 Scatterplot Program 1

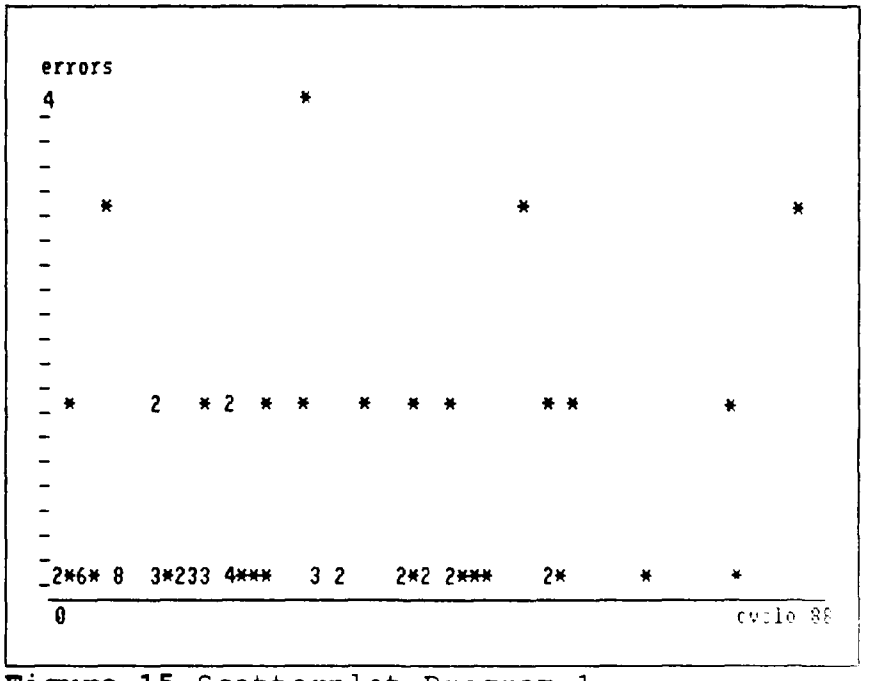

Figure 15 Scatterplot Program 1

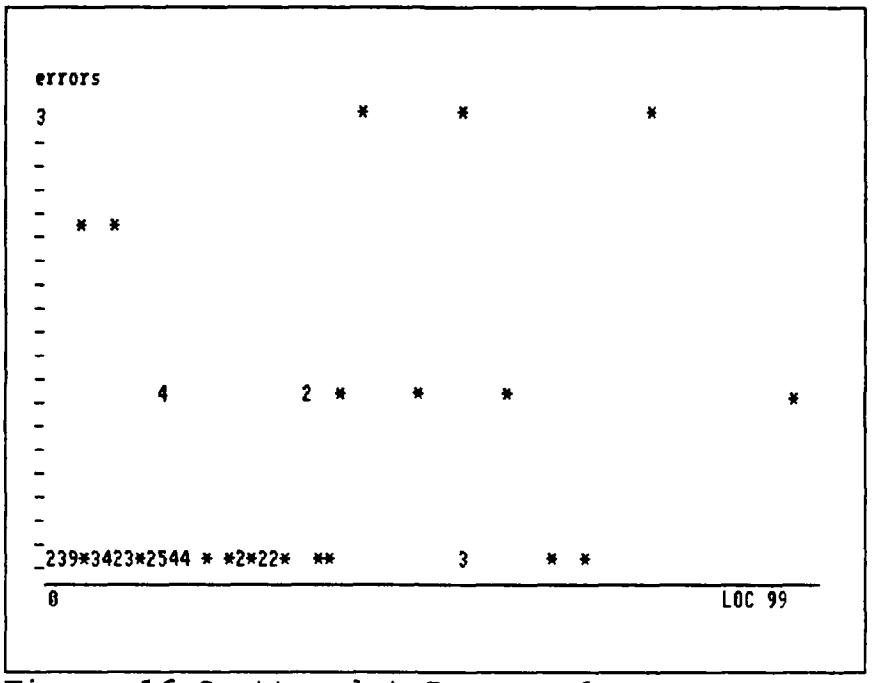

Figure 16 Scatterplot Program 1

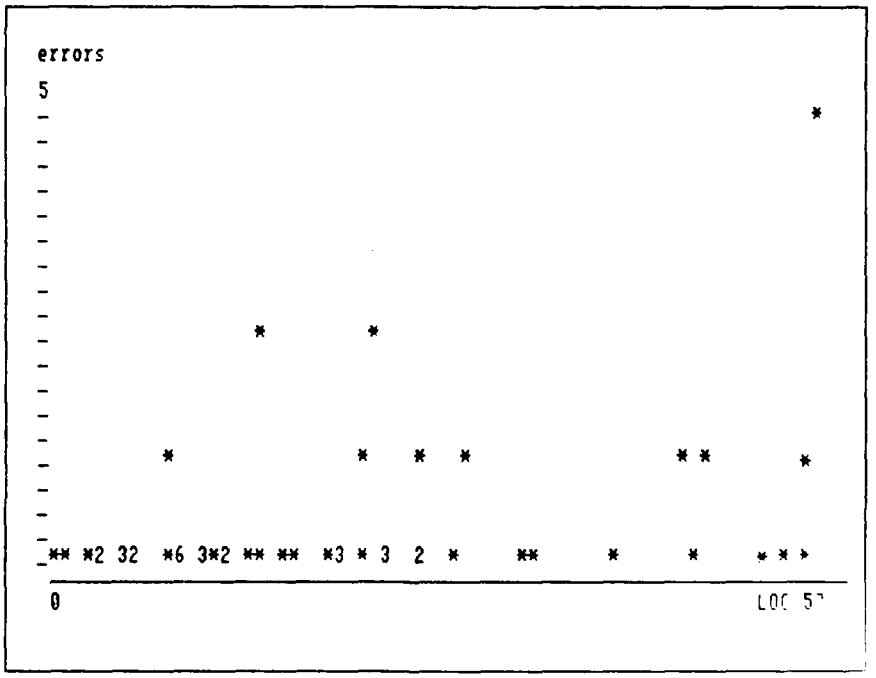

Figure 17 Scatterplot Program 1

There is probably not a linear relationship between LOC and faults, but inspection of the data indicated that the majority of the functions and procedures with faults have a size, in terms of LOC, greater than 30. There are however, a small number of procedures and functions where the Lines of Code measurement is less than 30 with the presence of faults.

b. Analysis of Observed Data

As with the statistical tests to determine if there was any relationship between the incidence of faults and the cyclomatic measure, the relationship between the measure of Lines of Code (LOC) and the incidence of faults was examined using the ANOVA and the Means Tests.

(1) ANOVA Tests. Functions and procedures for each program were first ordered according to LOC, from least to greatest, with each associated fault count. The data were divided approximately into thirds with respect to the total number of procedures because there were large numbers of procedures with the same cyclomatic complexity value, especially in the lower cyclomatic complexity value range.

The procedures and functions were divided into the following groups:

- bottom third with respect to the total of procedures, the mean fault count of which is designated:

  $\mu 1$

- middle third with respect to the total of procedures, the mean fault count of which is designated:

  $\mu 2$

- upper third with respect to the total of procedures, the mean fault count of which is designated:

  $\mu 3$

The null and alternative hypotheses in the one-factor ANOVA model are:

$Ho3: \mu 1 = \mu 2 = \mu 3$

Ha3: at least two of the population means are not equal.

Table IV provides a summary of the resultant ANOVA Test probabilities.

TABLE IV ANOVA LOC VS. FAULTS

Version	Probability
1	.00019740
2	.04930000
3	.00617900
4	.01770000
5	.08410000
6	.19170000
7	.04070000
8	.10230000

On the basis of the probabilities derived through the ANOVA tests, Ho3 can be rejected on the basis of five programs (one, two, three, four and seven), at the .05 alpha level. If the alpha tolerance level is broadened to .10, programs five and eight further support the rejection of Ho3. The ANOVA probability of program six is rather high and tends not to support the rejection of Ho3. The mean probability for all eight programs for the ANOVA tests is .0615 which does not support the rejection of Ho3 at the .05 alpha level. It should be noted, however, that the probability derived through the ANOVA test of program six is nearly two standard deviations (standard deviation = .0637) from the mean value (.0615) for all probabilities generated by the

ANOVA tests; as such it is an outlier. If the ANOVA probability of program six is discarded then the mean value probability for programs one, two, three, four, five, seven and eight is .0429 with a standard deviation of .0389. The mean value of this probability supports the rejection of Ho3 at the .05 level. In summary, if program six is considered an outlier, the ANOVA Tests do reject Ho3 and support Ha3, the hypothesis that the differences between the numbers of faults in the sub-groupings of procedures and functions are due to more than chance. Because, as was previously stated, the procedures and functions for each program were divided into 3 groups on the basis of LOC, it is suggested that there is a relationship between LOC and the incidence of faults, as was the case with cyclomatic complexity.

(2) Means Tests. A Means test was also performed on each of the eight programs in order to further substantiate the results of the ANOVA Tests. The t distribution was used in this test because the fault population standard deviation is unknown. The procedures and functions were ordered according to LOC, and were roughly divided in half according to the number of procedures and functions. This is entirely analogous to the division into thirds of the previous section.

The data are represented as follows:

- bottom half with respect to the total of procedures, the mean fault value of which is designated:

  $\sigma 1$

- upper half with respect to the total of procedures, the mean fault value of which is designated

  $\sigma 2$

The null and alternative hypotheses in the Means model are:

$Ho4: \sigma 1 = \sigma 2$

$Ha4: \sigma 1 \neq \sigma 2$

Table V provides a summary of the resultant Means Tests probabilities.

TABLE V MEANS LOC VS. FAULTS

Version	Probability
1	.00173800
2	.00932300
3	.00768200
4	.01250000
5	.01720000
6	.02750000
7	.06980000
8	.01160000

Ho4 can be rejected on the basis of seven programs (one, two, three, four, five, six, and eight), at the .05 alpha level. Additionally at the .07 alpha level program seven supports the rejection of Ho4. The mean value of all eight probability results from the Means Tests is .0197 with a standard deviation of .0216. In summary the Means Tests support the rejection of Ho4 and the acceptance of Ha4, the hypothesis that the differences between the numbers of faults in the sub-groupings of procedures and functions is due to more than chance. In other words, a relationship between higher measures of Lines of Code (LOC) and the increased incidence of faults is suggested by both the ANOVA Tests and the Means Tests.

c. Comparison of Cyclomatic Complexity and LOC as Predictors of The Incidence of Faults

Tables VI and VII provide a summary comparison of the results obtained from sections a and b. The column entitled **ANOVA Cyclo** (Table VI) lists the probabilities derived from application of the ANOVA Test to the programs in order to look at the faults as a function of the cyclomatic complexity. The column entitled **ANOVA LOC** (Table VI) lists the probabilities derived from application of the ANOVA Test to the programs in order to look at the faults as a function of the Lines of Code. The column entitled **Means**

Cyclo (Table VII) lists the probabilities derived from application of the Means Test to the programs in order to look at the faults as a function of the cyclomatic complexity. And lastly, the column entitled **Means LOC** (Table VII) lists the probabilities derived from application of the Means Test to the programs in order to look at the faults as a function of the Lines of Code.

TABLE VI SUMMARY OF ANOVA TESTS V(G) VS. FAULTS & LOC VS. FAULTS

Version	ANOVA Cyclo	ANOVA LOC
1	.00001308	.00019740
2	.10820000	.04930000
3	.00288300	.00617900
4	.02930000	.01770000
5	.10570000	.08410000
6	.37630000	.19170000
7	.00151100	.04070000
8	.12800000	.10230000

TABLE VII SUMMARY OF MEANS TESTS V(G) VS. FAULTS & LOC VS. FAULTS

Version	Means Cyclo	Means LOC
1	.00029840	.00173800
2	.02100000	.00932300
3	.02510000	.00768200
4	.06250000	.01250000
5	.02910000	.01720000
6	.08250000	.02750000
7	.02250000	.06980000
8	.04920000	.01160000

Inspection of the ANOVA tests (see table VI) shows that the probabilities were lower for the cyclomatic complexity than the LOC in three of eight the programs (one, three and seven) . In the other five programs (two, four, five, six and eight) the probabilities were lower for the LOC than the cyclomatic complexity, possibly indicating

a stronger relationship between size (LOC) and the incidence of faults. Inspection of the Means tests (see Table VII) shows that the ANOVA probabilities were lower for cyclomatic complexity as apposed to LOC in only two cases (one and seven).

The two columns of data shown in Table VI were compared using a Means test to determine if there existed a statistically significant difference between them. Similarly, a Means test was applied to the two columns of data in Table VII.

(1) Means test V(G) verses LOC for ANOVA Tests.

A Means Test was performed on the probability results derived from the ANOVA Tests on the two groups: faults as a function of cyclomatic complexity and faults as a function of LOC (see table VI). The means test produced a probability of .1921, which does not support the rejection of the equivalency of the two groups at any reasonable alpha level. In other words, it cannot be stated that there is a stronger relationship between cyclomatic complexity and faults than LOC and faults or vice versa with regard to the ANOVA Test probabilities.

(2) Means test V(G) verses LOC for Means Tests.

Additionally, a Means Test was conducted on the probability results derived from the Means Tests on the two groups: faults as a function of cyclomatic

complexity, and faults as a function of LOC (see table VI columns four and five). The means test produced a probability of .0854, which does not support the rejection of the equivalency of the two groups at the .05 alpha level.

d. Low LOC with Faults and Low V(G) with Faults

In order to further examine the relationship between cyclomatic complexity and LOC, the procedures and functions containing faults with low cyclomatic complexity or low LOC were examined to determine if there was a higher incidence of the other factor (higher LOC in the case of procedures or functions with low cyclomatic complexity and faults for example) to explain the faults.

(1) Statistical Tests to Examine Procedures with Low V(G) and faults. As previously stated, it was observed by inspection of the data that there appeared to be a higher incidence of faults in procedures and functions with a v(G) of roughly greater than 7. The Means tests and ANOVA tests support the contention that the differences in mean faults between low cyclomatic complexity procedures/functions and higher ones is due to more than chance. In this section, faulty procedures and functions with cyclomatic complexity values less than seven are examined to determine if there is any support for correlation between LOC and faults in these

procedures. The procedures and functions with a v(G) of less than seven were divided into two groups to compare LOC:

- faults present, the mean of LOC which is designated:

  $\sigma 1$

- no faults present, the mean of LOC which is designated:

  $\sigma 2$

The null and alternative hypotheses in the Means model are:

$Ho5: \sigma 1 = \sigma 2$

$Ha5: \sigma 1 \neq \sigma 2$

Table VIII provides a summary of the resultant Means Tests probabilities.

TABLE VIII MEANS LOW V(G) LOC VS. FAULTS

Version	Probability
1	.064000
2	.259900
3	.017200
4	.025300
5	.028900
6	.050800
7	.010400
8	.006272

Programs three, four, five, six, seven and eight support the rejection of Ho5 and the acceptance of Ha5 at the .05 alpha confidence level. Additionally if the alpha level is broadened to .10, program one supports the rejection of Ho5. The probability for program two is more than two standard deviations (.08) beyond the mean for all eight programs (.05) and as such is an outlier. The mean probability for all eight programs (.05) supports the rejection of Ho5 and the acceptance of Ha5 at the .05 alpha level. Inspection of the data revealed that in all programs the mean values of the LOC were higher in the groups with the presence of faults than in the group without them. In summary, there does appear to be support for a relationship between the faulty procedures and functions with a v(G) of less than seven and higher values for LOC.

(2) Statistical Tests to Examine Procedures with Low LOC and Faults. It was also determined upon inspection of the data and because of supporting results from Means tests and ANOVA tests that there was a higher incidence of faults in procedures and functions larger than 30 LOC. Similarly, as in the previous section, the Means Test was used to examine any correlation between v(G) and the incidence of faults in procedures with a size of less than 31 LOC. The procedures and functions with a Lines of

Code count (LOC) of less than 31 were divided into two groups to look at v(G):

- faults present, the mean value of v(G) which is designated:

  $\sigma 1$

- no faults present, the mean value of v(G) which is designated:

  $\sigma 2$

The null and alternative hypotheses in the Means model are:

$Ho6: \sigma 1 = \sigma 2$

$Ha6: \sigma 1 \neq \sigma 2$

Table IX provides a summary of the resultant Means Tests probabilities.

TABLE IX MEANS LOW LOC V(G) VS. FAULTS

Version	Probability
1	.085100
2	.364200
3	.308700
4	.477000
5	.111000
6	.181500
7	.160400
8	.218500

On the whole the resultant probabilities for all eight programs do not support the rejection of Ho6, with the possible exception of program 1. In other words there is virtually no support, in those procedures and functions with a LOC size of less than 31, for a relationship between faults and increased cyclomatic complexity.

(3) Means test comparison of probabilities from tables VIII and IX. In order to verify that there was no equivalency between the resultant probabilities obtained in subsections (1) and (2), a Means test was conducted on the probabilities from tables VIII and IX. The resultant probability was .003121, thus the equivalency of the two groups is rejected.

3. Summary of Results

In summary it was found, that with respect to the programs analyzed in this study, there does appear to be a correlation between the cyclomatic complexity and the incidence of faults. A correlation between the simple measure LOC and the incidence of faults was also found, but not significantly different from the correlation with cyclomatic complexity. Procedures and functions with a low cyclomatic complexity ($v(G) < 7$) that contained faults did exhibit a significant correlation between faults and LOC, but the converse was not true, in that small procedures and functions

(LOC < 31) with faults did not tend to have higherv(G) values than procedures without faults.

Lastly there is the question of the results of program six: in the case of the ANOVA test of faults with respect to cyclomatic complexity, six was the exception and as such did not exhibit a correlation between cyclomatic complexity and faults. Examination of this program in closer detail with respect to the location of faults has shown that, of those low v(G) procedures and functions with faults, 5 of the 11 total low v(G) faults were related to variable initialization or assignment, 2 were parameter passing faults and one was a calculation fault. In other words, just over 70% of the low end errors appeared to be related to complexity factors that are not directly linked to the incidence of decision nodes or logical branching (cyclomatic complexity).

III. SUMMARY AND CONCLUSIONS

A. SUMMARY

This study must be viewed in the context of the programs that were examined. The programs were developed by student coders in a university course environment, and are not necessarily representative of a commercial programming environment. Further, although the programs that were examined were thoroughly and rigorously debugged using various testing techniques (as indicated in chapter 2), and all eight versions essentially were subjected to the equivalent of unit testing, the programs were not further tested at a systems integration level of testing. These programs, being multiple versions of one design specification, simply are not suited to such an endeavor.

The results of this study have added to the general body of knowledge concerning the relationship of the cyclomatic complexity to the occurrence of faults. Firstly, this analysis of multiple versions of complex, relatively large programs (ranging in size from 1200 to 2400 LOC) indicates that there is a correlation between faults and the cyclomatic complexity measure. Further, it was found in this study that the results of the analysis of LOC to faults roughly parallels the relationship of cyclomatic complexity

to faults. It could not be determined in this context whether cyclomatic complexity or LOC had a stronger relationship to faults. Additionally, it would be interesting to undertake a similar study, based on the more thorough testing and accumulation of fault data, at a systems integration level of testing, and in a commercial environment.

B. RECOMMENDATIONS

With regard to the process of software testing, the cyclomatic complexity measure is supported as a predictor of potentially faulty regions of code, and as such it is a tool that the software manager could use to facilitate a more successful testing strategy and subsequent testing. On the other hand, because these results indicate that there is a correlation between larger modules and faults, and given that LOC is a very easily obtained metric, the software manager may be well advised to utilize LOC as opposed to cyclomatic complexity. A more cautious approach would, however, be the employment of both cyclomatic complexity and LOC as mutually supportive predictors of faulty code regions, as these two metrics both seem to parallel each other with regard to fault prediction.

C. OPEN RESEARCH QUESTIONS

There are several interesting research questions that have arisen in the course of this study which remain to be studied with regard to complexity issues. There is, for example, the issue of how to deal with cases where the cyclomatic complexity is not a predictor. Version six seems to fall into this category, considering the results of the ANOVA test. Another issue is how metrics can be developed to predict faults in relatively short (less than 30 LOC) and simple (in terms of cyclomatic complexity) modules. Intra-modular complexity is addressed by cyclomatic complexity. However, inter-modular complexity, which is potentially a rich source of faulty code, is not addressed by the cyclomatic complexity measure. These questions remain open to further research work.

LIST OF REFERENCES

1. Brooks, F.P., *The Mythical Man-month*, p. 8, Addison-Wesley Publishing Company, 1975.

2. McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, v. SE-2, no. 4, December 1976.

3. Boehm, B.W., TRW, "Improving Software Productivity," *IEEE Computer Magazine*, v. 20, no. 9, pp. 43-57, September 1987.

4. Beizer, B., "Software System Testing and quality Assurance," p. 14, *Van Nostrand Reinholder*, 1984.

5. Walsh, T.J., "A software reliability study using a complexity measure," *AFIPS Conference Proceedings*, v. 48, pp. 761-768, 1979.

6. Schneidewind, N.F., "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Enginerring, v. SE-5, no. 3, May 1979.*

7. Basili, V.R., Selby, R.W., Phillips, T., "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Transactions on Software Engineering*, v. SE-9, no. 6, pp. 652-663, November 1983.

8. Ward, W.T., "Software Defect Prevention Using McCabe's Complexity Metric," *Hewlett-Packard Journal*, v. 40, p. 64, April 1989.

9. Butler, L.P., "Software Quality Assurance Cyclomatic Complexity of a Computer Program," *Proceedings of the IEEE 1983 National Aerospace and Electronics Conference*, v. 2, pp. 867-73, 1983.

10. Meals, R.R., "An Experiment in the Implementation and Application of Halstead's and McCabe's Measures of Complexity," *Software Engineering Standards Application Workshop*, pp. 45-50, 1981.

11. Henry, S., Kafura, D., Harris K., "ON THE RELATIONSHIP AMONG THREE SOFTWARE METRICS," *Performance Evaluation Review*, v. 10, no. 1, pp. 3-10, 1981

12. Gollhofer, M., *Predicting Errors using McCabe's Metric*, Master's Thesis, University of California, Davis, 1983.

13. Myers, J.G., *The Art of Software Testing*, John Wyley & Sons, 1979.

14. Shimeall, T.J., *An Empirical Comparison of Software Fault Tolerance and Fault Elimination*, Ph.D. Disertation, U.C. Irvine, 1989.

APPENDIX

Program (parse1.c) is included as an appendix because it was developed specifically for this study as a tool designed to automatically calculate the cyclomatic complexity of procedures and functions in eight relatively large (1200-2400 LOC) Pascal programs. The use of such a tool greatly facilitates the determination of cyclomatic complexity (which is often referred to as a lexical metric), what would otherwise be a tedious and error prone activity. This is particularly true of this study, in which the programs analyzed were very complex in terms of the level of nesting, the depth of procedural scoping, and the length and complexity of constructs. The programming methodology employed in this lexical analyzer and parser is recursive descent.

```
/*program:            parse1.c
 *programmer:         Edwin A. Shuman
 *date last revision: 8 March 1990
 *computer:           Vax 11/785
 *compiler:           Berkeley C Compiler (4.3BSD)
 *
 *program description: This program is a pascal lexical analyzer and parser.
 *                    It is designed to read a pascal source file and
 *                    output a stream of tokens. Each token is comprised of
 *                    a "C" Structure containing the token itself
 *                    and its identification. The tokens are interpreted
 *                    according to the Backus-Naur Form for the Pascal
 *                    Language. On the basis of the grammatical contstructs
 *                    (specifically the Pascal Predicates: if-then, while,
 *                    repeat, case (conditions), for, or) the Cyclomatic
 *                    Complexity is derived.
 */

#include <ctype.h>
#include <ctype.h>
#include <stdio.h>


#define TRUE 1
#define FALSE 0


#define _and 257
#define _array 258
#define _begin 259
#define _case 260
#define _const 261
#define _div 262
#define _do 263
#define _downto 264
#define _else 265
#define _end 266
#define _file 267
#define _for 268
#define _function 269
#define _goto 270
#define _if 271
#define _in 272
#define _label 273
#define _mod 274
#define _nil 275
#define _not 276
#define _of 277
#define _or 278
#define _packed 279
#define _procedure 280
#define _program 281
#define _record 282
#define _repeat 283
#define _set 284
#define _then 285
#define _to 286
#define _type 287
#define _until 288
#define _var 290
#define _while 291
#define _with 292
#define _ident 293
#define _int 294
```

```
#define _real 295
#define _string 296
#define _assign 297
#define _plus 298
#define _adding_op 299
#define _divide 300
#define _leftpar 301
#define _rightpar 302
#define _colon 303
#define _semicol 304
#define _comma 305
#define _leftbrace 306
#define _rightbrace 307
#define _period 308
#define _pointer 309
#define _mult 310
#define _exp 311
#define _dotdot 312
#define _relational_op 313
#define _minus 314

/* reserve is an array of structures which holds all of the pascal
 * reserved words
 */

struct {
    char word[12];
    int value;
} reserve[] =

{           "and",          _and,
            "array",        _array,
            "begin",        _begin,
            "case",         _case,
            "const",        _const,
            "div",          _div,
            "do",           _do,
            "downto",       _downto,
            "else",         _else,
            "end",          _end,
            "file",         _file,
            "for",          _for,
            "function",     _function,
            "goto",         _goto,
            "if",           _if,
            "in",           _in,
            "label",        _label,
            "mod",          _mod,
            "nil",          _nil,
            "not",          _not,
            "of",           _of,
            "or",           _or,
            "packed",       _packed,
            "procedure",    _procedure,
            "program",      _program,
            "record",       _record,
            "repeat",       _repeat,
            "set",          _set,
            "then",         _then,
            "to",           _to,
            "type",         _type,
            "until",        _until,
            "var",          _var,
            "while",        _while,
```

```
        "with",             _with,
    "any_other",            _ident,
};

/* lexrec is a type definition of a structure which is the format of the token
 * returned by lex()
 */

typedef struct {
    char tok[15];
    int toktype;
} lexrec;

lexrec token, temptok, temptok2, temptok3, temptok4, temptok5;

lexrec stack[30];

int use_temp_tok, count;

static int line;
static int tokflag,tokflag1,tokflag2,tokflag3,tokflag4 = 0;
static int sc = -1;
/* lexdigit(c) is called if the standard input character is a digit */
/* returns token */

lexdigit(c)                                                    lexdigit
char c;

{
int d,
    i;
float value_dec;
    i = 0;
    while (isdigit(c)) {
        token.toktype = _int;
        token.tok[i] = c;
        c = getc(stdin);
        i++;
    }
    if (c == 'E') {
        token.tok[i] = c;
        token.toktype = _real;
        c = getc(stdin);
        i++;
        if ((c == '+') || (c == '-') || (isdigit(c))) {
            token.tok[i] = c;
            i++;
            c = getc(stdin);
            while (isdigit(c)) {
                token.tok[i] = c;
                c = getc(stdin);
                i++;
            }
        }
        ungetc(c,stdin);
        return;
    }
    if (c ==  ') {
        d = getc(stdin);
        if (isdigit(d)) {
            token.toktype = _real;
            token.tok[i] =  ;
            i++;
            token.tok[i] = d;
```

```
            c = getc(stdin);
            i++;
            while (isdigit(c)) {
                token.tok[i] = c;
                c = getc(stdin);
                i++;
            }
            if (c == 'E') {
                i++;
                token.tok[i] = c;
                c = getc(stdin);
                i++;
                if ((c == '+') || (c == '-') || (isdigit(c))) {
                    token.tok[i] = c;
                    i++;
                    c = getc(stdin);
                    while(isdigit(c)) {
                        token.tok[i] = c;
                        c = getc(stdin);
                        i++;
                    }
                }
            }
            ungetc(c,stdin);
            return;
        }
        else if (d == '.') {
            strcpy(temptok.tok, "..");
                temptok.toktype = _dotdot;
                use_temp_tok = TRUE;
        }
    }

    /* if the next char after the int was not 'E' or '.' */

    else {
        ungetc(c, stdin);
        return;
    }
}

/* lexalpha is called if the standard input is a letter or number */

lexalpha(c)                                                    lexalpha
char c;
{
int i;

    i = 0;
    while ((isalnum(c)) || (c == '_')) {
        token.tok[i] = c;
            c = getc(stdin);
            i++;
    }
    token.toktype = ident_token(token.tok);
    ungetc(c, stdin);
}

 * called if stdin is not a letter or a digit *
 * returns token *

lexswitch()                                                    lexswitch
char c;
{
```

```
int i,
    d,
    c;

    switch(c) {
        case \':                    /* a string */
                token.tok[0] = \';
                c = getc(stdin);
                i = 1;
                while (c != \') {
                        token.tok[i] = c;
                        c = getc(stdin);
                        i++;
                }
            c = getc(stdin);
            if (c == \') {
                c = getc(stdin);
                    while (c != \') {
                            token.tok[i] = c;
                            c = getc(stdin);
                            i++;
                    }
                    token.tok[i] = \';
                token.toktype = _string;
                    break;
            }
            else {
                ungetc(c,stdin);
                    token.tok[i] = \';
                token.toktype = _string;
                    break;
            }
            break;
        case '=':
            token.tok[0] = '=';
            token.toktype = _relational_op;
                break;
           case ')':
            token.tok[0] = ')';
            token.toktype = _rightpar;
                break;
           case ':':
            c = getc(stdin);
            if (c == '=') {
                strcpy(token.tok, ":=");
                token.toktype = _assign;
            }
            else {

                    token.tok[0] = ':';
                token.toktype = _colon;
                ungetc(c,stdin);
            }
                break;
        case ';':
                token.tok[0] = ';';
            token.toktype = _semicol;
                break;
        case ',':
                token.tok[0] = ',';
            token.toktype = _comma;
                break;
        case '[':
                token.tok[0] = '[';
```

```
        token.toktype = _leftbrace;
            break;
    case ']':
            token.tok[0] = ']';
        token.toktype = _rightbrace;
            break;
    case '.':
            token.tok[0] = '.';
        token.toktype = _period;
            break;
    case '^':
            token.tok[0] = '^';
        token.toktype = _pointer;
            break;

       case '/':
        token.tok[0] = '/';
            token.toktype = _divide;
        break;
    case '+':
            token.tok[0] = '+';
        token.toktype = _plus;
            break;
    case '-':
        token.tok[0] = '-';
        token.toktype = _minus;
        break;
    case '*':
            if (c != d) {
             token.tok[0] = '*';
                token.toktype = _mult;
        }
        break;
    case '<':
        d = getc(stdin);
        if (d == '=') {
             strcpy(token.tok, "<=");
             token.toktype = _relational_op;
        }
        else if (d == '>') {
             strcpy(token.tok, "<>");
             token.toktype = _relational_op;
        }
        else {
            strcpy(token.tok, "<");
            token.toktype = _relational_op;
            ungetc(d,stdin);
        }
        break;

    case '>':
        d = getc(stdin);
        if (d == '=') {
             strcpy(token.tok, ">=");
             token.toktype = _relational_op;
        }
        else {
             c = getc(stdin);
             if (c == '=') {
                  strcpy(token.tok, ">=");
                  token.toktype = _relational_op;
             }
             else {
                  strcpy(token.tok, ">");
```

```
                    token.toktype = _relational_op;
                    ungetc(e,stdin);
                    ungetc(d,stdin);
                }
            }
            break;
    }
}

/* ident_token(t) compares the token to the list of pascal reserved words
 * returning the reserved word identification if found
 */
```

ident_token

```
ident_token(t)
char *t;
{
int i,
    f,
    found,
    i = 0;

    found = FALSE;
    while (!found && strcmp(reserve[i].word, "any_other")) {
        if (!strcmp (reserve[i].word, t))
            found = TRUE;
        else
            i++;
    }
    return (reserve[i].value);
}
/* token_set() sets all of the elements of the array token.tok which is part of  * the structure of a token to 0
 */
```

token_set

```
token_set() {
int i;
    for (i = 0; i < 15; i++) {
        token.tok[i] = '\0';
        token.toktype = _goto;
    }
    return;
}


lexrec lex() {
int    c, d;

static int  leftbraceflag = 0;

    if (use_temp_tok) {
        use_temp_tok = 0;
            return(temptok);
    }
    else {
        c = getc(stdin);
        token_set();

        c=c;            while (c == ' ' || c == '\n' || c == '\t') {
            if (c == '\n') {
                /* printf("line is # %d\n", line); */
                line++;
            }
            c = getc(stdin);
        }
```

```
cont1:  if (c == '(') {
            d = getc(stdin);
        if (d == '*') {
                c = getc(stdin);
            while (c != '*') {
                if (c == '\n') {
                    line++;
                }
                        c = getc(stdin);
            }
            c = getc(stdin);  /* rightpar */

            c = getc(stdin);
            if ((c == ' ') || (c == '\n') || (c == '\t'))
                goto cont;
        }
        else {
            token.tok[0] = '(';
            token.toktype = _leftpar;
                ungetc(d, stdin);
            return(token);
        }
    }
       if(c == '{') {
        while (c != '}') {
                c = getc(stdin);
            if (c == '\n') {
                /* printf("line is # %d\n", line); */
                line++;
            }
        }
        c = getc(stdin);
        if ((c == ' ') || (c == '\n') || (c == '\t'))
            goto cont;
        if (c == '(')
            goto cont1;
    }


    if (isdigit(c)) {
            /* printf("lexdigit(%d %c)\n",c,c); */
            lexdigit(c);
       }
    else {
         * printf("1 %d-%c\n",c,c); *
        if (isalpha(c)) {
        /* printf("lexalpha(%d %c)\n",c,c); */
            lexalpha(c);
                }
        else {
         * printf("lexswitch(%d %c)\n",c,c); *
                lexswitch(c);
                }
        }
* printf("token=%d %s\n",token.toktype,token.tok); *
    return(token);

   }
}

* Lex is the parser section of the program.  Each function is entitled *
```

```
/* to correspond to the BNF name of the Pascal Construct which it handles */

/* after called must get next token */
program_heading() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _ident) {
        sc++;
        strcpy(stack[sc].tok, token.tok);
            token = lex();
            if (token.toktype == _leftpar) {
                toker = lex();
                if (token.toktype == _ident) {
                        token = lex();
                        while (token.toktype == _comma) {
                            token = lex();
                            if (token.toktype == _ident)
                          token = lex();
                 }
                        if (token.toktype == _rightpar) {
                            token = lex();
                            if (token.toktype == _semicol)
                          return;
                          else return;
                        }
                    else return;
                 }
                    else return;
                }
                else return;
     }
     else return;
}
```

program_heading

```
/* after called must get next token */
label_dec_part() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _int) {
        token = lex();
        while (token.toktype != _semicol) {
            token = lex();
            if (token.toktype == _int)
                token =lex();
            else return;
        }
        return;
    }
    else return;
}

constant() {
    if ((token.toktype == _int) || (token.toktype == _real))
        return;
    else if ((token.toktype == _plus) || (token.toktype == _minus)) {
        token = lex();
        if ((token.toktype == _int) || (token.toktype == _real))
            return;
    }
```

label_dec_part

constant

```
    else if (token.toktype == _ident)
        return;
    else if ((token.toktype == _plus) || (token.toktype == _minus)) {
        token = lex();
        if (token.toktype == _ident)
            return;
    }
    else if (token.toktype == _string)
        return;

    else
        return;
}
/* after called must get next token */
const_def() {                                                    const_def
    if (token.toktype == _ident)
        token = lex();
    else return;
    if (token.toktype == _relational_op)
        token = lex();
    else return;
    constant();
}

/* sets tokflag before terminates */
const_def_part() {                                               const_def_part
    if(!tokflag)
        token = lex();
    if(tokflag)
        tokflag = FALSE;
    const_def();
    token = lex();
    while (token.toktype == _semicol) {
        token = lex();
        if(token.toktype == _ident) {
            const_def();
            token = lex();
        }
        else  {
            tokflag = TRUE;
            return;
        }
    }
}


type_definition_part() {                                         type_definition_part
    if(!tokflag)
        token = lex();
    if(tokflag)
        tokflag = FALSE;
    type_definition();
    if (!tokflag4)
        token = lex();
    tokflag4 = FALSE;
     * type_definition->type->structured_type -> *
     * unpacked_structure_type->set_set_type -> *
     * simple_type (simple_type gets next token) *

    while (token.toktype == _semicol) {
        token = lex();
        if (token.toktype == _ident) {
            type_definition();
```

```
            if (!tokflag4)
                token =lex();
            if (tokflag4)
                tokflag4 = FALSE;
        }
        else {
            tokflag = TRUE;
            return;
        }
    }
}
```

type_definition

```
type_definition() {
    if (token.toktype == _ident) {
        token = lex();
        if (token.toktype == _relational_op) {
            token = lex();
                type();
        }
        return;
    }
    return;
}
```

scalar_type

```
scalar_type() {

    if (token.toktype == _ident) {
            token = lex();
            if (token.toktype == _comma) {
                while (token.toktype != _rightpar) {
                        token = lex();
                }
                token = lex();
                    return;
                }
        return;
    }
    return;
}
```

subrange_type

```
subrange_type() {

    constant();
    token = lex();
    if (token.toktype == _dotdot) {
        token = lex();
            constant();
    }
}
```

simple_type

```
simple_type() {
    if (token.toktype == _leftpar) {
        token = lex();
        scalar_type();
        tokflag4 = TRUE;
    }
    else if (token.toktype == _ident) {
            token = lex();
        if (token.toktype == _dotdot) {
            token = lex();
            constant();
            token =lex();
            tokflag4 = TRUE;
        }
```

```
        else
            tokflag4 = TRUE;
    }
    else if ((token.toktype == _int) || (token.toktype == _real) || (token.toktype ==_minus) || (token.toktype == _plus) || (token.toktype
        subrange_type();
        token = lex();
        tokflag4 = TRUE;
    }
}

unpacked_structured_type() {                                    unpacked_structured_type

    if (token.toktype == _array)
        array_type();
    else if (token.toktype == _record)
        record_type();
    else if (token.toktype == _set)
        set_type();
    else if (token.toktype == _file)
        file_type();
    else return;
}

type() {                                                        type
    /* simple_type checks to see if the  token.toktype is _leftpar first */
    if ((token.toktype != _pointer) && (token.toktype != _packed) && (token.toktype != _array) && (token.toktype != _record) && (t
        simple_type();
    if ((token.toktype == _packed) || (token.toktype == _array) || (token.toktype == _record) || (token.toktype == _set) || (token.toktype
        structured_type();
    }
    if (token.toktype == _pointer)
        pointer_type();
}


array_type() {                                                  array_type
        token = lex();
    if (token.toktype == _leftbrace) {
        token =lex();
        while (token.toktype != _rightbrace)
            token = lex();
        token = lex();
        if (token.toktype == _of) {
            token = lex();
            type();
        }
    }
}


structured_type() {                                             structured_type
    unpacked_structured_type();
    if (token.toktype == _packed) {
        token = lex();
        unpacked_structured_type();
    }

}


record_type() {                                                 record_type
    if (!tokflag)
        token = lex();
    tokflag = FALSE;
```

```
    field_list();
    if (token.toktype == _end) {
        token = lex();
        tokflag4 = TRUE;
    }
}
```

field_list

```
field_list() {

    if (token.toktype == _case) {
            variant_part();
        return;
    }
    else if (token.toktype == _ident) {
        fixed_part();
        if (token.toktype == _case)
            variant_part();
        else return; /* token.toktype == _end */
    }
    else return;
}
```

fixed_part

```
fixed_part() {

    record_section();
    if (!tokflag4) {
        token = lex();
    /* record_section->type->structured_type->      */
    /* unpacked_structure_type->if_set_type ->      */
    /* simple_type (simple_type gets next token)    */
    }
    tokflag4 = FALSE;
    while (token.toktype == _semicol) {
            token = lex();
        if ((token.toktype != _end) && (token.toktype != _case)){
                record_section();
            if (!tokflag4)
                    token = lex();
            tokflag4 = FALSE;
        }
        else return;
    }
}
```

record_section

```
record_section() {

    if (token.toktype == _ident) {
            token = lex();
            while (token.toktype == _comma) {
                token = lex();
            if (token.toktype = _ident)
                    token = lex();
        }
        if (token.toktype == _colon) {
            token = lex();
                type();
        }
    }
}
```

variant_part

```
variant_part() {
```

```
    token = lex();
    if (token.toktype == _ident) {
            token = lex();
        /* tag field */
            if (token.toktype == _colon) {
                token = lex();
                if (token.toktype == _ident) {
                 token = lex();
                       if (token.toktype == _of) {
                            token = lex();
                            variant();
                            token = lex();
                            while (token.toktype == _semicol) {
                           token = lex();
                           if (token.toktype == _end) {
                              return;
                           }
                                   variant();
                                   token = lex();
                            }
                            tokflag = TRUE;
                            return;
                 }
                }
            }
            else if (token.toktype == _of) {
             token = lex();
                variant();
                token = lex();
                while (token.toktype == _semicol) {
                 token = lex();
                 if (token.toktype == _end) {
                     return;
                 }
                 else {
                            variant();
                            token = lex();
                 }
                }
                tokflag = TRUE;
                return;
        }
    }
    return;
}

variant() {                                                    variant

    case_label_list();
    if (token.toktype == _colon) {
        token = lex();
        if (token.toktype == _leftpar) {
            token = lex();
            field_list();
                if (token.toktype == _rightpar)
                    return;
        }
    }
}


set_type() {                                                   set_type
    if (!tokflag)
        token = lex();
```

```
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _of) {
        token = lex();
        simple_type();
    }
}
```

file_type

```
file_type() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _of) {
        token = lex();
        type();
    }
}
```

pointer_type

```
pointer_type() {
    if (token.toktype == _pointer)
        token = lex();
    if (token.toktype == _ident) {
        token = lex();
        tokflag4 = TRUE;
    }
}
```

variable_declaration_part

```
variable_declaration_part() {
    token = lex();
    variable_declaration();
    while(token.toktype == _semicol) {
        token = lex();
        if (token.toktype == _ident)
                variable_declaration();
        else {      /* must return to main */
            tokflag = TRUE;
            return;
        }
    }
    /* variable_declaration_type->structured_type->unpacked_structure_type-> */
    /* if set_type->simple_type (simple_type gets next token)                 */
}
```

variable_declaration

```
variable_declaration() {
    if (token.toktype == _ident) {
        token = lex();
            while (token.toktype == _comma) {
            token = lex();
            if (token.toktype == _ident)
                token = lex();
        }
        if (token.toktype == _colon) {
            token =lex();
                type();
        }
        tokflag4 = TRUE;
    }
}
```

procedure_heading

```
procedure_heading() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _ident) {
        sc++;
        strcpy (stack[sc].tok,token.tok);
        token = lex();
        if (token.toktype == _semicol)
            return;
            /* return to calling function */
        else if (token.toktype == _leftpar) {
            formal_parameter_section();
            while (token.toktype == _semicol) {
                formal_parameter_section();
            }
            if (token.toktype == _rightpar)
                token = lex();
            if (token.toktype == _semicol)
                return;
        }
    }
}
```

formal_parameter_section

```
/* formal_parameter_section gets next token */
formal_parameter_section() {
    token = lex();
    if (token.toktype == _ident)
        parameter_group();
        /* gets next token */
    else if (token.toktype == _var) {
            token = lex();
            parameter_group();
        /* gets next token */
    }
    else if (token.toktype == _function) {
            token = lex();
            parameter_group();
        /* gets next token */
    }
    else if (token.toktype == _procedure) {
            token = lex();
            if (token.toktype == _ident) {
                token = lex();
                while (token.toktype == _comma) {
                        token = lex();
                        if (token.toktype == _ident)
                            token = lex();
                }
             * when finished gets next token - rightpar or semicol */
        }
    }
}
```

parameter_group

```
/* gets next token either a semicol or rightpar */
parameter_group() {

    if (token.toktype == _ident) {
        token = lex();
```

```
        while (token.toktype == _comma) {
                token = lex();
                if (token.toktype == _ident)
                 token = lex();
        }
        if (token.toktype == _colon) {
                token = lex();
                if (token.toktype == _ident)
                 token = lex();
        }
    }
    return;
}
```

function_heading

```
function_heading() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _ident) {
        sc++;
        strcpy(stack[sc].tok,token.tok);
        token = lex();
        if (token.toktype == _colon) {
            token = lex();
                if (token.toktype == _ident) {
                    token = lex();
                    if (token.toktype == _semicol) {
                     token = lex();
                     tokflag = TRUE;
                     return;
                    }
                }
                return;
        }
        else {
            if (token.toktype == _leftpar) {
                formal_parameter_section();
                    while (token.toktype == _semicol) {
                        token = lex();
                        formal_parameter_section();
                    }
                    if (token.toktype == _rightpar) {
                     token = lex();
                     if (token.toktype == _colon) {
                         token = lex();
                         if (token.toktype == _ident) {
                             token = lex();
                             if (token.toktype == _semicol) {
                                 token = lex ();
                                 tokflag = TRUE;
                                     return;
                             }
                         }
                     }
                    }
            }
        }
    }
}
```

statement

```
/* <unlabelled statement> or <label> : <unlabelled statement> */
statement() {
    tokflag = FALSE;
    tokflag1=tokflag2=tokflag3=tokflag4 = FALSE;

    if (token.toktype == _int) {
            token = lex();
        if (token.toktype == _colon) {
                token = lex();
                unlabelled_statement();
        }
    }
    unlabelled_statement();
}
```

unlabelled_statement

```
/* <simple statement> or <structured statement> */
unlabelled_statement() {
    if ((token.toktype == _begin) || (token.toktype == _if) || (token.toktype == _case) || (token.toktype == _while) || (token.toktype ==
        structured_statement();
    else
        simple_statement();
}
```

simple_statement

```
/* simple statement is <assignment statement> or <procedure statement> or */
/* goto statement or empty statement */
simple_statement() {
    if(token.toktype == _ident) {
        temptok2 = token;
        token = lex();
        temptok3 = token;
        tokflag1 = TRUE;
        if ((token.toktype != _assign) && (token.toktype != _leftbrace) && (token.toktype != _period) && (token.toktype != _pointer)
            procedure_statement();
        else
            if (token.toktype != _semicol)
                assignment_statement();
            else
                tokflag = TRUE;
    }
    else if (token.toktype == _goto)
        go_to_statement();
    else
        tokflag = TRUE;
}
```

variable

```
/* when variable() is finished executing, tokflag is set because variable goes */ /* one token beyond */
variable() {
    if (tokflag1) {
        if (temptok2.toktype == _ident) {
            if ((temptok3.toktype == _leftbrace) || (temptok3.toktype == _period) || (temptok3.toktype == _pointer)) {
                if (temptok3.toktype == _pointer) {
                    temptok3.toktype = _goto;
                        token = lex();
                    tokflag = TRUE;
                }
                else if (temptok3.toktype == _leftbrace) {
                    temptok3.toktype = _goto;
                        token = lex();
```

```
                    expression();
                    while (token.toktype == _comma) {
                        token = lex();
                        expression();
                    }
                if (token.toktype == _rightbrace) {
                    token = lex();
                    tokflag = TRUE;
                }
            }
            else if (temptok3.toktype == _period) {
                temptok3.toktype = _goto;
                    token = lex();
                    if (token.toktype == _ident)
                        token = lex();
                    tokflag = TRUE;
            }
        }
        /*else
            tokflag5 = TRUE; */
           while ((token.toktype == _leftbrace) || (token.toktype == _period) || (token.toktype == _pointer)) {
            if (token.toktype == _pointer) {
                    token = lex();
                tokflag = TRUE;
            }
            else if (token.toktype == _leftbrace) {
                    token = lex();
                    expression();
                    while (token.toktype == _comma) {
                        token = lex();
                        expression();
                    }
                if (token.toktype == _rightbrace) {
                    token = lex();
                    tokflag = TRUE;
                }
            }
            else if (token.toktype == _period) {
                    token = lex();
                    if (token.toktype == _ident) {
                      token = lex();
                    tokflag = TRUE;
                }
                    else return "error";
            }
        }
    }
    temptok2.toktype = _goto;
    tokflag1 = FALSE;
}

/* tokflag1 = TRUE, condition not met */
else {
    if (temptok4.toktype == _ident) {
            if (temptok5.toktype == _leftbrace) || (temptok5.toktype == _period) || (temptok5.toktype == _pointer)) {
            if (temptok5.toktype == _pointer) {
                temptok5.toktype = _goto;
                    token = lex();
                tokflag = TRUE;
            }
            else if (temptok5.toktype == _leftbrace) {
                temptok5.toktype = _goto;
                    token = lex();
                    expression();
```

```
                    while (token.toktype == _comma) {
                        token = lex();
                        expression();
                    }
                if (token.toktype == _rightbrace) {
                    token = lex();
                    tokflag = TRUE;
                }
            }
            else if (temptok5.toktype == _period) {
                temptok5.toktype = _goto;
                    token = lex();
                    if (token.toktype == _ident) {
                      token = lex();
                    tokflag = TRUE;
                }
            }
        }
        temptok4.toktype = _goto;
        temptok5.toktype = _goto;
            while ((token.toktype == _leftbrace) || (token.toktype == _period) || (token.toktype == _pointer)) {
            if (token.toktype == _pointer) {
                    token = lex();
                tokflag = TRUE;
            }
            else if (token.toktype == _leftbrace) {
                    token = lex();
                      expression();
                      while (token.toktype == _comma) {
                          token = lex();
                          expression();
                      }
                if (token.toktype == _rightbrace) {
                    token = lex();
                    tokflag = TRUE;
                }
            }
            else if (token.toktype == _period) {
                    token = lex();
                    if (token.toktype == _ident) {
                      token = lex();
                    tokflag = TRUE;
                }
            }
        }
    }

    else if (token.toktype == _ident) {
        token = lex();
        tokflag = TRUE;
            while ((token.toktype == _leftbrace) || (token.toktype == _period) || (token.toktype == _pointer)) {
            if (token.toktype == _pointer) {
                    token = lex();
                tokflag = TRUE;
            }
            else if (token.toktype == _leftbrace) {
                    token = lex();
                      expression();
                      while (token.toktype == _comma) {
                          token = lex();
                     tokflag = FALSE;
                          expression();
                      }
                if (token.toktype == _rightbrace)
```

```
                    token = lex();
                    tokflag = TRUE;
                }
                else if (token.toktype == _period) {
                    token = lex();
                      if (token.toktype == _ident) {
                        token = lex();
                      tokflag = TRUE;
                    }
                }
            }
        }
    }
} /* end variable */
```

assignment_statement

```
assignment_statement() {
    variable();
    if (token.toktype == _assign) {
            token = lex();
            expression();
    }
}
```

indexed_variable

```
indexed_variable() {

    expression();
    while (token.toktype == _comma) {
            token = lex();
            expression();
    }
    if (token.toktype == _rightbrace)
            return;

}
```

expression

```
/* expression goes one token beyond because of simple_expression */
expression() {
    simple_expression();
    if ((token.toktype == _relational_op) || (token.toktype == _in)) {
        token = lex();
        simple_expression();
    }
    tokflag = TRUE;
}
```

relational_op

```
relational_op() {

    if ((token.toktype == _in) || (token.toktype == _relational_op))
        return;

}
```

simple_expression

```
simple_expression() {
    if (tokflag2) {
        if ((temptok4.toktype == _minus) || (temptok4.toktype == _plus)) {
                term();
            while ((token.toktype == _plus) || (token.toktype == _minus) || (token.toktype == _or)) {
```

```
                if (token.toktype == _or)
                    count++;
                token = lex();
                term();
                }
        }
        else {
                term();
                while ((token.toktype == _minus) || (token.toktype == _plus) || (token.toktype == _or)) {
                if (token.toktype == _or)
                    count++;
                token = lex();
                term();
                }
        }
        tokflag2 = FALSE;
    }
    /* not tokflag2 */
    else {
        if ((token.toktype == _minus) || (token.toktype == _plus) || (token.toktype == _or)) {
                token = lex();
                term();
            while ((token.toktype == _minus) || (token.toktype == _plus) || (token.toktype == _or)) {
                if (token.toktype == _or)
                    count++;
                token = lex();
                term();
                }
        }
        else {
                term();
                while ((token.toktype == _minus) || (token.toktype == _plus) || (token.toktype == _or)) {
                if (token.toktype == _or)
                    count++;
                token = lex();
                term();
                }
        }
    }
}
```

```
/* after term has been called next token already called */
term() {                                                                    term
    factor();
    if (!tokflag)
        token = lex();
    tokflag = FALSE;
    while ((token.toktype == _mult) || (token.toktype == _divide) || (token.toktype == _mod) || (token.toktype == _and) || (token.toktyp
            token = lex();
            factor();
    }
}
```

```
/* factor := <variable> or <unsigned constant> or ( <expression> ) */
/* <function designator> or <set> or not <factor> */
factor() {                                                                  factor
    if (token.toktype == _not) {
        token = lex();
            while (token.toktype == _not)
              token = lex();
```

```
        if (token.toktype == _leftpar) {
                token = lex();
                expression();
                if (token.toktype == _rightpar)
                    token = lex();
        }
        if (token.toktype == _leftbrace)
            set();
        if (token.toktype == _ident) {
            function_designator();
            variable();
            tokflag = TRUE;
        }
        unsigned_constant();
    }
    else {
        if (token.toktype == _leftpar) {
                token = lex();
                expression();
                if (token.toktype == _rightpar)
                    token = lex();
        }
        if (token.toktype == _leftbrace)
            set();
        if (token.toktype == _ident) {
            function_designator();
            if (token.toktype != _semicol)
                variable();
            tokflag = TRUE;
        }
        unsigned_constant();
    }
}
```

unsigned_constant

```
unsigned_constant() {
    if (tokflag2) {
        if ((temptok4.toktype == _int) || (temptok4.toktype == _real))
            return;
        if ((temptok4.toktype == _string) || (temptok4.toktype == _ident) || (temptok4.toktype == _nil))
                return;
    }
    else {
        if ((token.toktype == _int) || (token.toktype == _real)) {
            token = lex();
            tokflag = TRUE;
            return;
        }
        if ((token.toktype == _string) || (token.toktype == _ident) || (token.toktype == _nil)) {
            token = lex();
            tokflag = TRUE;
                return;
        }
    }
}
```

function_designator

```
function_designator() {
    if (token.toktype == _ident) {
        temptok2 = token;
        token = lex();
        temptok3 = token;
```

```
        if (token.toktype == _leftpar) {
                token = lex();
                actual_parameter();
            while (token.toktype == _comma) {
                    token = lex();
                 actual_parameter();
            }
            if (token.toktype == _rightpar)
                token = lex();
                /* to get semicol recognized by compound_statement */
        }
        tokflag1 = TRUE;
    }
}
```

set

```
set() {
    if (token.toktype == _leftbrace) {
            token = lex();
            element_list();
            /* token = lex(); */
            if (token.toktype == _rightbrace) {
             token = lex();
                tokflag = TRUE;
        }
            else return;
    }
}
```

element_list

```
/* tokflag set to true if not a multielement element list */
element_list() {
    if (token.toktype != _rightbrace) {
        element();
        if (!tokflag)
            token = lex();
        tokflag = FALSE;
        while (token.toktype == _comma) {
                token = lex();
                element();
            if (!tokflag)
                token = lex();
            tokflag = FALSE;
        }
        if (token.toktype != _comma) {
            return
        }
    }
}
```

element

```
 * if token after expression is not _dotdot set tokflag so that *
 * next function called uses current token *
element() {
    expression();
    if (token.toktype == _dotdot) {
        token = lex();
            expression();
    }
    tokflag = TRUE;
}
```

procedure_statement

```
procedure_statement() {
    if (token.toktype == _leftpar) {
        token = lex();
            actual_parameter();
        if(!tokflag)
                token = lex();
        if(tokflag)
            tokflag = FALSE;
            while (token.toktype == _comma) {
                token = lex();
                actual_parameter();
            if(!tokflag)
                 token = lex();
            if(tokflag)
                 tokflag = FALSE;
            }
            if (token.toktype == _rightpar) {
             token = lex();
             tokflag = TRUE;
                return;
        }
    }
}
```

actual_parameter

```
actual_parameter() {
        /* at this point left_par has already been called */
        /* temptok4 = token; */
        token = lex();
        temptok5 = token;
        if ((token.toktype == _leftbrace) || (token.toktype == _period) || (token.toktype == _pointer)) {
            token = lex();
            variable();
        }
        else {
            tokflag2 = TRUE; */
            expression();
        /* ) */
    /* ) */
    /* if procedure_identifier or function_identifier, */
    /* token.toktype == _ident) is handled by variable */
}
```

go_to_statement

```
go_to_statement() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    if (token.toktype == _int)
        return;
}
```

structured_statement

```
/* <compound statement> or <conditional statement> or <repetitive statement> */
/* or <with statement> */

structured_statement() {
    if (token.toktype == _begin)
        compound_statement();
    if ((token.toktype == _if) || (token.toktype == _case))
        conditional_statement();
```

```
    if ((token.toktype == _while) || (token.toktype == _repeat) || (token.toktype == _for))
        repetitive_statement();
    else
        if (token.toktype == _with)
            with_statement();
}
```

compound_statement

```
compound_statement() {
    token = lex();
    statement();
    /* statement may go one token past because of simple_expression */
    if (!tokflag)
        token = lex();
    tokflag = FALSE;
    while (token.toktype == _semicol) {
        token = lex();
        if (token.toktype == _end) {
            token = lex();
            tokflag = TRUE;
                return;
        }
        else {
                statement();
            if (!tokflag) {
                token = lex();
                tokflag = FALSE;
            }
        }
  }
    if (token.toktype == _end) {
        token = lex();
        tokflag = TRUE;
        return;
    }
}
```

conditional_statement

```
conditional_statement() {
    if (token.toktype == _if)
        if_statement();
    else if (token.toktype == _case)
        case_statement();
}
```

if_statement

```
if_statement() {
    count++;
        token = lex();
    expression();
    if (token.toktype != _then)
        token = lex();
    if (token.toktype == _then) {
        token = lex();
        statement();
        if (token.toktype == _end) {
            return
        }
            if (token.toktype == _else) {
            token = lex();
                statement();
```

```
            return;
        }
        else
        return;
    }
    printf("In if, count=%d\n",count);
}
```

case_statement

```
case_statement() {
    if (!tokflag)
        token = lex();
    if (tokflag)
        tokflag = FALSE;
    expression();
    if (token.toktype == _of) {
        token = lex();
        if ((token.toktype == _plus) || (token.toktype == _minus) || (token.toktype == _int) || (token.toktype == _string) || (token.toktyp
            case_list_element();
            count++;
            if (!tokflag)
                token = lex();
            if (tokflag)
                tokflag = FALSE;
                while (token.toktype == _semicol) {
                    token = lex();
                    case_list_element();
                count++;
                if (!tokflag)
                    token = lex();
                if (tokflag)
                    tokflag = FALSE;
                }
                if (token.toktype == _end) {
                token = lex();
                tokflag = TRUE;
                return;
            }
        }
        else {
                if (token.toktype == _end) {
                token = lex();
                tokflag = TRUE;
                return;
            }
        }
    }
}
```

case_list_element

```
case_list_element() {
    case_label_list();
    if (token.toktype == _colon) {
        token = lex();
        if (token.toktype == _semicol)
            return;
        else
            statement();
    }
}
```

case_label_list

```
case_label_list() {
```

```
    constant();
    token = lex();
    while (token.toktype == _comma) {
        token = lex();
            constant();
        token = lex();
    }
    return;
}
```

repetitive_statement

```
repetitive_statement() {
    if (token.toktype == _while)
        while_statement();
    if (token.toktype == _repeat)
        repeat_statement();
    if (token.toktype == _for)
        for_statement();
    return;
}
```

while_statement

```
while_statement() {
   count++;
   if (!tokflag)
           token = lex();
   if (tokflag)
        tokflag = FALSE;
   expression();
   if (token.toktype == _do) {
       token = lex();
       statement();
   }
}
```

repeat_statement

```
repeat_statement() {
    count++;
    token = lex();
    statement();
    if (!tokflag)
        token = lex();
    tokflag = FALSE;
    while (token.toktype == _semicol) {
        token = lex();
        statement();
        if (!tokflag)
            token = lex();
        if (tokflag)
            tokflag = FALSE;
   }
   if (token.toktype == _until) {
        token = lex();
        expression();
   }
   return
}
```

for_statement

```
for_statement() {
```

```
    count++;
    token = lex();
    if (token.toktype == _ident) {
        token = lex();
            if (token.toktype == _assign) {
                token = lex();
                for_list();
                if (token.toktype == _do) {
                 token = lex();
                     statement();
                 return;

                }
        }
    }
}
```

for_list

```
for_list() {

    expression();
    if ((token.toktype == _to) || (token.toktype == _downto)) {
            token = lex();
            expression();
    }
    tokflag = TRUE;
    return;
}
```

with_statement

```
with_statement() {
    if (!tokflag)
            token = lex();
    if (tokflag)
        tokflag = FALSE;
    record_variable_list();
    if (token.toktype == _do) {
        token = lex();
            statement();
    }
}
```

record_variable_list

```
record_variable_list() {
    variable();
        if (!tokflag)
            token = lex();
        tokflag = FALSE;
    while (token.toktype == _comma) {
        if (!tokflag)
                token = lex();
        tokflag = FALSE;
            variable();
    }
}
```

main

```
main() {
int startline,
    linecount,

    while(!feof(stdin)) {
        if(!tokflag)
            token = lex();
```

```
if(tokflag)
    tokflag = FALSE;
if (token.toktype == _program)
    program_heading();
   else if (token.toktype == _label)
    label_dec_part();
else if (token.toktype == _type)
    type_definition_part();
else if (token.toktype == _const)
    const_def_part();
   else if (token.toktype == _packed){
    printf("Have packed\n");
    structured_type();
    }
else if (token.toktype == _array){
    printf("Have array\n");
    array_type();
    }
   else if (token.toktype == _record){
    printf("Have record\n");
    record_type();
    }
   else if (token.toktype == _case){
    printf("Have case\n");
    case_statement();
    }
   else if (token.toktype == _set){
    printf("Have set\n");
    set_type();
    }
else if (token.toktype == _file){
    printf("Have file\n");
    file_type();
    }
   else if (token.toktype == _var)
    variable_declaration_part();
   else if (token.toktype == _procedure)
    procedure_heading();
   else if (token.toktype == _function)
    function_heading();
   else if (token.toktype == _begin) {
    startline = line;
    count = 0;
    compound_statement();
    if (sc >= 0) {
        linecount = line - startline;
        printf("****%s:    %d    %d\n", stack[sc].tok,count+1,linecount);
        sc -= 1;
    }
}
   else if (token.toktype == _if){
    printf("Have if\n");
    if_statement();
    }
   else if (token.toktype == _while){
    printf("Have while\n");
    while_statement();
    }
   else if (token.toktype == _repeat){
    printf("Have repeat\n");
    repeat_statement();
    }
   else if (token.toktype == _for){
    printf("Have for\n");
```

```
            for_statement();
            }
        else if (token.toktype == _with) {
            printf("Have with\n");
            with_statement();
    }
    else if (token.toktype == _pointer) {
        printf("Have pointer\n");
            pointer_type();
    }
        else if (token.toktype == _goto)
            go_to_statement();
}
}
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Timothy J. Shimeall Department of Computer Science, Code 52SM Naval Postgraduate School Monterey, California 93943-5002	5
4.	William Haga Department of Administrative Sciences, Code AS/Hg Naval Postgraduate School Monterey, California 93943-5002	1
5.	LT Edwin A. Shuman IV Department Head Class 113 Surface Warfare Officers School Command Newport, Rhode Island 02841-5012	1
6.	Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5002	1
7.	Chairman, Code 54 Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943-5002	1

← END