

## **INFORMATION TO USERS**

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# U·M·I

University Microfilms International  
A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
313:761-4700 800:521-0600



**Order Number 9333104**

**The interaction of computer program debugging tools, field dependence, and computer programming languages in higher education computer language courses**

**Laverty, Joseph P., Jr., Ed.D.**

**University of Pittsburgh, 1993**

**Copyright ©1993 by Laverty, Joseph P., Jr. All rights reserved.**

**U·M·I**  
300 N. Zeeb Rd.  
Ann Arbor, MI 48106



**THE INTERACTION OF COMPUTER PROGRAM DEBUGGING TOOLS,  
FIELD DEPENDENCE, AND COMPUTER PROGRAMMING LANGUAGES  
IN HIGHER EDUCATION COMPUTER LANGUAGE COURSES**

By

Joseph P. Lavery, Jr.

B.A. Duquesne University

M.B.A. Duquesene University

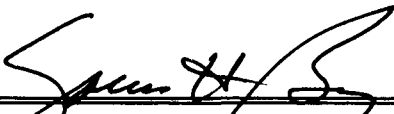
Submitted to the Graduate Faculty in the School  
of Education in partial fulfillment of  
the requirements for the degree of  
Doctor of Philosophy

University of Pittsburgh

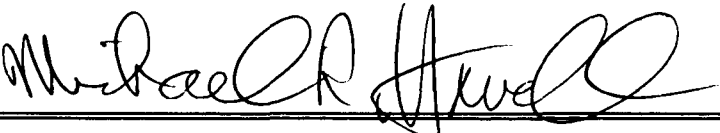
1993

FACULTY ENDORSEMENT AND FINAL REVIEW COMMITTEE

Faculty Member

  
\_\_\_\_\_  
Dr. Louis H. Berry, Research Advisor  
University of Pittsburgh

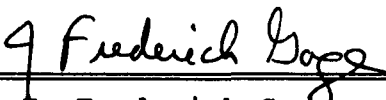
School of Education

  
\_\_\_\_\_  
Dr. Michael R. Harwell  
University of Pittsburgh

School of Education

  
\_\_\_\_\_  
Dr. Barbara A. Seels  
University of Pittsburgh

School of Education

  
\_\_\_\_\_  
Dr. J. Frederick Gage  
University of Pittsburgh

School of Education

Date: 4/16/93

Copyright by Joseph P. Lavery

1993

LAB

**THE INTERACTION OF COMPUTER PROGRAM DEBUGGING TOOLS,  
FIELD DEPENDENCE, AND COMPUTER PROGRAMMING LANGUAGES  
IN HIGHER EDUCATION COMPUTER LANGUAGE COURSES**

Joseph P. Lavery, Ph.D.

University of Pittsburgh, 1993

This study explores the relationship between the interactive and traditional program debugging tools, field dependence, programming language and a student programmer's ability to locate and correct logic errors in BASIC and COBOL computer programs. The field dependence construct identifies field independent and field dependent individuals by differentiating the techniques they use to understand an element within the context of a given problem. Research into field dependence strongly suggests that this construct may effect the ability of an individual to perform a program debugging task. It would appear that the use of the interactive program debugger tool may provide a supplantation function in facilitating the debugging process for field dependent programmers.

Two intact groups of COBOL programming students (n=40) and two intact groups of BASIC programming students (n=45) were randomly selected and assigned to either the interactive or traditional debugging treatments. Assignment of subjects as field dependent, field independent, or indeterminate was based upon



their performance on the Group Embedded Figures Test. A program debugging posttest, consisting of five programs, was administered on a computer to students to measure the student's ability and the amount of time to locate and correct program logic errors. The total percentage score from three programming prerequisite tests was used as a covariate in this study.

The findings from this study may be summarized as follows:

- 1) There were no statistically significant effects or interactions between the student's ability or time to correct program logic errors, and the use of program debugging tools, field dependence or programming languages.
- 2) Programming prerequisite skills were significantly related to the student's ability or time to correct program logic errors.

Based upon the results of this study the following recommendations can be made: (a) the use of traditional program debugging tools is an adequate instructional tool and (b) computer programming curricula should be designed to ensure that students master syntactical and algorithmic programming skills.

## **ACKNOWLEDGEMENTS**

The author would like to express appreciation to the members of his doctoral committee:

Dr. Louis Berry, Chairman, for providing a strong foundation in psychology of human learning and his detailed contributions to my summary of the findings and recommendations for further research. His personal encouragement to help me improve my research writing-style will provide benefits beyond this study.

Dr. Michael Harwell for providing a thorough review and his on-going counsel in the design of this study and statistical proof of the findings. These detailed contributions has made this study a very special learning experience for the author.

Dr. Barbara Seels for her detailed review, comments and constructive suggestions covering my search of the literature and other topics.

Dr. J. Fred Gage for his detailed review and insights provided by his experience in computer programming. These comments ensure that the findings of this study would be of practical significance for computer programming educators.

Special acknowledgement is expressed to my colleagues, both faculty and staff, at Robert Morris College. Their reviews, comments and suggestions ensured a comprehensive and quality research study.

To my father and Rosie thank you for your support. To my children: Patrick, Jonathan, Ian, Shawn and Micki, I would like to thank them for their understanding and patience. I will love them always.

Special thanks to my wife and best friend, Karen, for her continuous support and love over the years that enabled me to accomplish this personal goal in life.

As a final note, I would like to remember two individuals who are no longer with us to enjoy this accomplishment. Dr. William Faith, Ph.D., University of Pittsburgh, will always be remembered as a researcher and excellent educator. Hopefully, I can continue this fine family tradition.

To my mother, Helen Jackson Laverty, I wish she was here to share this accomplishment, the love of her grandchildren and mine.

## TABLE OF CONTENTS

	Page
ABSTRACT . . . . .	iv
ACKNOWLEDGEMENTS . . . . .	vi
LIST OF TABLES AND FIGURES . . . . .	xi
I. INTRODUCTION	
A. Importance of Computer Programming Education in Higher Education . . .	1
B. Possible Advantages of the Interactive Program Debugger . . . .	8
C. Purpose of the Study . . . . .	10
D. Delimitations and Assumptions . . .	10
E. Research Questions . . . . .	12
F. Definitions of Terms . . . . .	13
II. REVIEW OF LITERATURE	
A. Introduction . . . . .	16
B. Computer Program Debugging . . . .	17
C. Problem Solving and Short Term Memory . . . . .	23
D. Cognitive Style and Programming . . . . .	25
E. Learner Control . . . . .	32
F. Orientation to the Study . . . . .	34

	Page
<b>III. RESEARCH METHODOLOGY</b>	
A. Problem Restated . . . . .	38
B. Research Design . . . . .	38
C. Methodological Limitations . . . . .	41
C. Subjects . . . . .	46
D. Instructional Materials . . . . .	51
E. Procedures . . . . .	53
F. Instruments . . . . .	56
G. Data Analysis . . . . .	64
<b>IV. RESEARCH FINDINGS</b>	
A. Introduction . . . . .	74
B. Demographic Profile . . . . .	75
C. Descriptive Statistics	
1. Prerequisite Tests . . . . .	87
2. Program Debugging Posttest . . . . .	91
D. Selection of the Covariate . . . . .	102
E. Analysis of Covariance . . . . .	105
F. Secondary Analysis . . . . .	112
<b>V. Summary and Research Findings</b>	
A. Summary . . . . .	114
B. Discussion of Findings . . . . .	116
C. Conclusions . . . . .	123
B. Further Research Recommendations	128
 <b>BIBLIOGRAPHY . . . . .</b>	 <b>140</b>

**APPENDICES**

<b>A. Prerequisite COBOL Knowledge . . . .</b>	<b>156</b>
<b>B. Prerequisite BASIC Knowledge . . . .</b>	<b>158</b>
<b>C. Program Syntax and Execution Errors .</b>	<b>160</b>
<b>D. CI201 Business Programming . . . . .</b>	<b>161</b>
<b>E. CS4/007 BASIC Programming . . . . .</b>	<b>164</b>
<b>F. Pre-Course Student Survey . . . . .</b>	<b>167</b>
<b>G. BASIC and COBOL Student Activity Journals . . . . .</b>	<b>170</b>
<b>H. COBOL Debugging Posttest . . . . .</b>	<b>172</b>
<b>I. BASIC Debugging Posttest . . . . .</b>	<b>200</b>
<b>J. Interactive Debugger Illustrations. .</b>	<b>222</b>

LIST OF TABLES AND FIGURES

<u>Table Number</u>	<u>Description/Title</u>	<u>Page Number</u>
1	Rating of the Interactive Debugging Program Control Options	4
2	Rating of the Interactive Debugging Program Inspection Options	5
3	Rating of the Interactive Debugging Break Options	5
4	Rating of the Interactive Debugging Program Sequence Options	6
5	Instructional Comparison of Two Interactive Program Debuggers	6
6	Possible Cognitive Advantages of Interactive Program Debuggers	8
7	Psychological Difficulties in Program Debugging	18
8	Traditional Program Debugging Techniques	19
9	Alternative Instructional Formats used in the Study	47
10	Schedule of Surveys and Tests used in the Study	54
11	Weighting Scale used to Score the Debugging Posttest	63
12	List of Variables and their Coding Schemes used in the Study	64
13	Sex of the Participants Classified by Instructional Sections	75
14	Age of the Participants Classified by Instructional Sections	75
15	College Credits Taken by Participants prior to Study Classified by Instructional Sections	76

<u>Table Number</u>	<u>Description/Title</u>	<u>Page Number</u>
16	Participants Classified by Collegiate Year and Instructional Section	76
17	Participants Classified by Major and Instructional Section	77
18	Prior Computer Programming and Computer Interest Classified by Instructional Section	78
19	Prior Microcomputer Experience (in months) of the Participants Classified by Instructional Section -- Descriptive Statistics	79
20	Prior Microcomputer Usage Experience Classified by Type of Microcomputer, Apple, Macintosh, IBM Compatible, and other	80
21	Number of Computer Programming Language Courses taken by the Participant Prior to Study Classified by Instructional Section -- Descriptive Statistics	80
22	Previous Courses in Computer Programming Language Classified by Programming and Instructional Section -- BASIC, COBOL, PASCAL, and other	81
23	Prior Microcomputer Software Experience -- Spreadsheet, Word Processing, and Other	82
24	Prior Home and Computer Usage Experience -- Spreadsheet, Word Processing, Programming	83
25	Group Embedded Figures Test Results Classified by Instructional Section -- Descriptive Statistics	84



<u>Table Number</u>	<u>Description/Title</u>	<u>Page Number</u>
26	Number of Field Independents, Field Dependents and Indeterminate as measured by the Group Embedded Figures Test Classified by Instructional Section	85
27	Hours Worked per Week by the Participant during Study Classified by Instructional Section -- Descriptive Statistics	86
28	Hours Worked per Week by the Participant during Study Classified by Instructional Section -- Descriptive Statistics	86
29	Analysis of the Programming Language Prerequisite Tests Classified by Programming Language -- Descriptive Statistics, KR-20, Point Biserial and Difficulty Index Analysis	88
30	Analysis of Total Prerequisite Test Score (PREREQ) Classified by Instructional Section -- Descriptive Statistics	89
31	Comparison of Individual Prerequisite Test Score Means Classified by Instructional Section	89
32	Prerequisite Test 1 Percentage Classified by Instructional Section -- Descriptive Statistics	90
33	Prerequisite Test 2 Percentage Classified by Instructional Section -- Descriptive Statistics	90
34	Prerequisite Test 3 Percentage Classified by Instructional Section -- Descriptive Statistics	90
35	Posttest Program Debugging Score (LOCCOR) Classified by Instructional Section -- Descriptive Statistics	91

<u>Table Number</u>	<u>Description/Title</u>	<u>Page Number</u>
36	Posttest Program Debugging Time (TIMECOR) Classified by Instructional Section -- Descriptive Statistics	92
37	Posttest Program Mean Debugging Score Classified by Instructional Section and Test Question	94
38	Posttest Program Mean Debugging Time Classified by Instructional Section and Test Question	94
39	Posttest Program Debugging Score Classified by Test Question -- Descriptive Statistics, Difficulty Index, Discrimination Index, Point Biserial	95
40	Analysis of Debugging Tool used Posttest Classified by Test Question for SECT1 (BASIC Traditional Treatment Group) -- Descriptive Statistics	98
41	Analysis of Debugging Tool used Posttest Classified by Test Question for SECT3 (BASIC Interactive Treatment Group) -- Descriptive Statistics	99
42	Analysis of Debugging Tool used Posttest Classified by Test Question for SECT2 (COBOL Traditional Treatment Group) -- Descriptive Statistics	100
43	Analysis of Debugging Tool used Posttest Classified by Test Question for SECT2 (COBOL Interactive Treatment Group) -- Descriptive Statistics	101
44	Pearson Correlation Analysis of PROGEXP (prior programming experience) and LOCCOR, TIMECOR and PREREQ and associated p values	103

<u>Table Number</u>	<u>Description/Title</u>	<u>Page Number</u>
45	Pearson Correlation Analysis of PREREQ (total percentage prerequisite test scores) and LOCCOR, TIMECOR and PROGEXP and associated p values	103
46	2 x 3 x 2 GLM Analysis of LOCCOR (the ability to locate and correct a program logic error) -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	107
47	2 x 3 x 2 Factorial GLM-type Analysis of Variance for LOCCOR (the ability to locate and correct a program logic error) -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	107
48	2 x 3 x 2 Factorial GLM-type Analysis of Covariance for LOCCOR (the ability to locate and correct a program logic error) -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	108
49	2 x 3 Factorial GLM-type Analysis of Variance for TIMECOR (the time to locate and correct a program logic error) for the BASIC programming language -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	110

<u>Table Number</u>	<u>Description/Title</u>	<u>Page Number</u>
50	2 x 3 Factorial GLM-type Analysis of Covariance for TIMECOR (the time to locate and correct a program logic error) for the BASIC programming language -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	111
51	2 x 3 Factorial GLM-type Analysis of Variance for TIMECOR (the time to locate and correct a program logic error) for the COBOL programming language -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	111
52	2 x 3 Factorial GLM-type Analysis of Covariance for TIMECOR (the time to locate and correct a program logic error) for the COBOL programming language -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	111
53	2 x 3 x 3 Factorial GLM-type Analysis of Covariance for Debugging Posttest Score for the Fourth Question -- GLM summary statistics, degrees of freedom, source of variation, sum of squares, means squared and associated F values	113

<u>Figure Number</u>	<u>Description/Title</u>	<u>Page Number</u>
1	Research Design - A 2 X 3 x 2 Factorial Analysis of the Ability to Locate a Computer Program Logic Error	39
2	Research Design - A 2 X 3 Factorial Analysis of the Time to Locate a Logic Error in a COBOL Program	39
3	Research Design - A 2 X 3 Factorial Analysis of the Time to Locate a Logic Error in a BASIC Program	40

## INTRODUCTION TO THE STUDY

The importance of computer programming education at all levels of education has continued to grow (Tenner, 1984). While the programming languages COBOL, C and PASCAL predominate in post secondary education, there has been increasing interest in LOGO and BASIC instruction at the elementary and secondary levels (Maddux, 1989, Papert, 1980). The Occupational Outlook Handbook (1990) views academic computer programming curricula as a career path. Other educators have viewed academic computer programming as a tool to help students develop problem solving skills (Martin & Hearne, 1990; McCoy & Dowl, 1989).

There has been considerable research into the role of the computer as an instructional tool to supplement various traditional curricula ( e.g., Computer Based Instruction, Hypertext). However, there has been little research into the use of the computer as an instructional tool in computer programming curricula. While other educational disciplines have successfully applied the computer as an instructional tool, computer programming educators have been reluctant to use computer technology as a tool to improve the quality of instruction in their curriculum (Lavery, 1990). It is ironic that computer educators who provide students with educational opportunities in computer technologies often

fail to see the value of computer technology as an instructional tool in their own curriculum.

Historically, computer programming has provided post-secondary students with many career paths. The Occupational Outlook Handbook (1990) indicates that careers in computer programming will increase at an annual rate of 53% over the next ten years. Two other computer related career areas, system analysts and computer peripheral operators, are expected to grow at a rate of 59% and 29%, respectively. These and other related computer career areas generally require some type of programming background.

Knowledge of computer programming also can provide career benefits to noncomputer majors. Kutscher (1990) suggested that students preparing for careers in the areas of mathematics, accounting, finance and operations research, would benefit from a course in computer programming. Many colleges and universities have required computer programming courses for various noncomputer majors.

A computer program language curriculum may offer benefits beyond career opportunities. Research has indicated that computer language programming may transfer problem solving skills to other disciplines. Developing strategies, planning, logical thinking, manipulation of variables and debugging are skills that have been used historically to describe the process of problem solving (McCoy & Dodl, 1989). In computer programming language curricula, these

problem solving concepts are presented through the topics of algorithmic development, flowcharting, string and numeric variables, and syntax and execution debugging skills.

Computer programming curricula have also served educators by introducing computer technology into the classroom. Educators have, for years, recognized the importance of the computer as an instructional tool in the classroom (Jackson, 1986). Many schools are turning to the computer programming curriculum as a low cost alternative to a comprehensive computer literacy policy (Maddux, 1989).

While interest in computer programming education has increased, changes in computer technology have provided the computer programming educator with a new instructional tool, the interactive computer program debugger. The interactive program debugger permits the student programmer to watch his/her program execute line by line, step by step, in the native code of the programming language. With an interactive computer program debugger, student programmers can learn how to code by watching the visual animation of the execution of their program. Through the use of the interactive computer program debugger student programmers are able to peer into the mystique of an executing program, thereby providing opportunities to locate and correct many program logic errors. Tables 1 thru 4 summarize some of the interactive debugging options available to programmers. Table 5 presents



a summarized instructional comparison of the MicroFocus  
COBOL and MicroSoft QuickBASIC interactive program debugger.

Table 1			
Interactive Debugging Control Options (implementation and rating)			
Type of Control	Options	COBOL	BASIC
Student Control of Program Execution	Line by Line Student Program Control (Visual Animation of Program Code)	Excel. (Step)	Excel. (F8)
	Procedure by Procedure Student Program Control	Poor (Zoom/Break)	Excel. (F10)
	Automatic Line Stepping with Speed Control (Visual Animation of Program Code)	Excel. (GO)	N.A.
	Full Speed Execution with Animated Break Option	Excel. (Zoom)	Excel. (F5)
	Structured Chart Animation of Program Code	Excel.	N.A.

<b>Table 2</b>			
<b>Interactive Debugging Inspection Options (implementation and rating)</b>			
<b>Type of Control</b>	<b>Options</b>	<b>COBOL</b>	<b>BASIC</b>
<b>Inspection and Manipulation of Data Contents</b>	<b>Single Variable Inspection</b>	<b>Excel. Query</b>	<b>Poor Watch Var.</b>
	<b>Multiple Variable Inspection</b>	<b>Excel. Adv. Query</b>	<b>Poor Watch Var.</b>
	<b>Interactive Change of Variable Contents during program Execution</b>	<b>Excel. Monitor</b>	<b>N.A.</b>

<b>Table 3</b>			
<b>Interactive Debugging Break Options (implementation and rating)</b>			
<b>Type of Control</b>	<b>Options</b>	<b>COBOL</b>	<b>BASIC</b>
<b>Setting Interactive Break Points</b>	<b>Unconditional Break Points</b>	<b>Excel. Break</b>	<b>Excel. F9</b>
	<b>Conditional Break Points</b>	<b>N.A.</b>	<b>Excel. by Menu</b>

<b>Table 4</b>			
<b>Interactive Debugging Sequence Options (implementation and rating)</b>			
<b>Type of Control</b>	<b>Options</b>	<b>COBOL</b>	<b>BASIC</b>
<b>Interactive Alteration of the Sequence of Execution</b>	<b>Change the Sequence of Execution</b>	<b>Good RESET</b>	<b>N.A.</b>
	<b>Execute Instructions not included in Original Code during Program Execution</b>	<b>Good DO</b>	<b>Excel. Immed. Mode</b>

<b>Table 5</b>		
<b>Overall Instructional Comparisons of two Interactive Program Debuggers</b>		
<b>Quality of Instructional Delivery</b>	<b>COBOL</b>	<b>BASIC</b>
<b>Ease of Use</b>	<b>Excel.</b>	<b>Good</b>
<b>Use of Color as an Instructional Cue</b>	<b>Excel.</b>	<b>Excel.</b>
<b>Use of Graphical, i.e., boxes, underlines as an Instructional Cue</b>	<b>Excel.</b>	<b>Poor</b>
<b>Use of a Graphical Animation Techniques</b>	<b>Excel.</b>	<b>N.A.</b>
<b>Interface with Program Editor</b>	<b>Good.</b>	<b>Excel.</b>

In the past many student programmers have used the "black box" approach (Pressman, 1987, p. 470). This program debugging strategy would require programmers to submit their program and input data to the computer (the black box) and

then compare the actual program outputs to the expected program outputs. Backtracking and hand-tracing the program's source code commonly supplemented the "black box" approach.

Program flowcharting and other graphical tools have also been used in the computer programming curricula as a planning and program tool. These graphical images were used to graphically represent the internal algorithmic processes of the computer program. However, these graphical images only provided an indirect symbolic relationship between actual program source code.

Writing a computer program and debugging computer program errors are demanding cognitive tasks. Some researchers have estimated that 50% of the program development effort is spent on testing, finding and correcting logic and execution errors (Bell & Pugh, 1987; Ward, 1988; Yourdan & Constantine, 1979). Though computer program writing (coding a program) and program debugging may appear procedurally interrelated, they may represent two different, complex sets of cognitive activities.

Popular use of manual tracing strategies suggests that cognitive processing demands of program debugging may exceed the storage and retrieval constraints of short term memory. This may suggest that the matrix of program operations and manipulated data elements may quickly exceed Miller's (1956) seven information units even in simplistic programs.

Interactive program debuggers may offer the advantage of releasing short term memory resources for the storage and processing of other programming debugging tasks. This reduction in cognitive overhead may permit the student programmer to better perceive the program bug, retrieve solutions from long term memory and develop better problem solving strategies. Table 6 summarizes some of the possible cognitive instructional advantages offered by interactive program debuggers.

<b>Table 6</b>	
<b>Possible Cognitive Advantages of Interactive Program Debuggers</b>	
1.	Student control of program execution may increase the ability of the student programmer to detect a program error and to restructure a solution to correct the program error.
2.	The ability to watch and track variable contents, and to automate the execution of a program may reduce various debugging clerical tasks and decrease the demand on short term memory resources.
3.	Various color and graphical visual queuing techniques that may direct and maintain a student programmer's concentration.
4.	Textual and graphical animation tools may provide contextual organizational tools that may aide field dependent programmers to study a specific program logic errors.
5.	Color, student control and textual animation tools may increase the programmer's motivation, interest and contribute to elaboration and long term memory storage.

The interactive program debugger may enable the student to visualize the execution of their program through the use of text and graphical images. This visual constructive process may benefit some students more than others. Jesky & Berry (1991) state: "Research into the use of visuals for instructional purposes has increasingly identified the interaction between an individual learner's cognitive skills and the design factors incorporated in the instructional method" (p. 290). The fact that field dependent individuals have trouble disembedding objects from their context and structuring information to develop solutions (Messick, 1977) suggests that field dependent students may benefit more from an interactive program debugger.

While there has been some research concerning the programming productivity of professional programmers, there has been little research into the processes that students use to debug logic and execution errors in their programs. Most programming language texts provide little discussion of debugging techniques and strategies in relationship to the time students and professional programmers spend debugging logic and execution errors in programs (Benander & Benander, 1989). Yet, it is debugging logic or execution errors in a program that creates the most fear and anxiety in students (Shneiderman, 1980).

Interactive program debuggers are not a recent phenomena. During the early 1980's many mainframe computer

companies offered interactive program debuggers for their family of program language translators. In spite of their increased availability, there appears to be no research into the use of interactive program debuggers in computer program language education or professional settings.

Studying the value of interactive computer program debuggers may lead to better instructional strategies and may balance the cognitive demands of computer programming curricula. A key, but as yet unstudied question, is whether computer program interactive debuggers will help students to learn program debugging skills faster, better and with less effort.

#### Purpose of the Study

The purpose of this study was to evaluate the effects of computer program debugging tools, computer program languages, and field dependence on the ability of a student programmer to locate and correct logic errors in a computer program.

#### Delimitations

This study will involve students enrolled in two sections of an introductory COBOL programming course and two sections of an introductory BASIC programming course. Each COBOL and BASIC course involved with the study will cover three credit

hours for one 15 week semester. Each course involved in this study will be conducted in the same semester at the same school. All programming courses will be conducted by the same instructor.

The COBOL and BASIC programming languages were chosen for study rather than other programming languages because these languages: (a) represented the largest population to be studied, (b) were required courses for noncomputer majors, (c) contrasted the effects of an interpretive versus a compiled language, and (d) offered the most developed, student-accessible program debuggers.

The results of this study will be limited to entry level COBOL and BASIC programming curricula in higher education. While topics concerning the process of program development are discussed in these curricula, the results of this study are limited to the tasks of debugging logic errors in COBOL and BASIC programs during the instructional process. Furthermore, computer debugging tasks studied will be limited to the five debugging tasks presented in the posttest program and these results may not be applicable to other program debugging tasks.

### Assumptions

The academic experience of this researcher in computer programming curricula significantly aided in the development of the research design and test materials of this study. In



addition, this experience was valuable in the analysis of the data and the interpretation of the results.

### Research Questions

1. Do college level students in entry level COBOL and BASIC programming courses who use interactive program debugging techniques differ in their ability to locate and correct a logic error in a syntax and execution error-free program from other college level programming students who use a traditional program debugging technique?
2. Does field dependence of college level students in entry level COBOL and BASIC programming courses differentially affect their ability to locate and correct a logic error in a syntax and execution error-free program using an interactive program debugger versus a traditional program debugging technique?
3. Do college level students in entry level COBOL or BASIC programming courses who use interactive program debugging techniques differ in the amount of time required to locate and correct a logic error in a syntax and execution error-free from other college level programming students who use a traditional program debugging technique?

## Definition of Terms

### Entry level college level programming courses

Students enrolled in a college entry level COBOL or BASIC programming class.

### Ability to locate and correct logic errors

Scores obtained by students on a computer debugging posttest (Appendices J & K), which requires students to locate and correct the following logic errors in a syntax and execution error-free program: (a) failure to execute a statement within a loop, (b) incorrect execution sequence, and (c) the incorrect execution of conditional statements.

### Syntax and execution error-free program

COBOL and BASIC program source code that contains no syntax or execution errors (Appendix C).

### Traditional debugging technique

Traditional debugging technique is the procedure of embedding CRT output statements (e.g. DISPLAY (COBOL) or PRINT (BASIC)) within a source program and testing the program by the reexecution of the program and review of the output results.

### Interactive program debugger techniques

Interactive program debugging tools developed by MicroFocus, Inc. (COBOL) and MicroSoft, Inc. (QuickBASIC). These two interactive program debugging tools are comparable in terms of performance, capabilities and instructional time.

### Prior programming experience

The number of computer programming courses taken prior to enrollment in the course. High school or college, informal self-instruction programming course work and professional programming experience were included.

### Prerequisite COBOL and BASIC knowledge

Scores obtained by students from three objective achievement tests, which will measure the student's ability to develop, code, compile, and correct syntax errors in a source program and apply various program algorithms, e.g., accumulation, counting, and high/low. Appendix A lists prerequisite COBOL knowledge necessary for a student to debug logic errors. Appendix B lists prerequisite BASIC knowledge necessary for a student to debug logic errors.

Field dependence/field independence

". . . refers to a consistent mode of approaching the environment in analytical as opposed to global terms. It denotes the ability to articulate figures as discrete from their backgrounds and an ability from disembedding contexts . . ." (Messick, 1977, p. 14). The field independent pole is a mode of perception in which individuals perceive the surrounding environment analytically. Field dependent individuals, on the other hand, are more effected by the surrounding environment and perceive things less analytically (Witkin, Goodenough & Oltman, 1979).

For the purpose of this study this construct will be represented by the score obtained by students from the Group Embedded Figures Test (Witkin, et al., 1971). Individuals with low scores on the GEFT test tend to be considered field dependent, while individuals who score high on the GEFT test are considered field independent (Witkin, et al., 1971).

## REVIEW OF RELATED RESEARCH AND LITERATURE

### Introduction

Direct research into computer programming language instruction and student program debugging has been limited. While previous research in student program debugging has been lacking, recent cognitive research findings have provided evidence to suggest that the computer program debuggers may be both an effective instructional tool, as well as a cognitive tool. Developing a computer program and debugging computer program errors are demanding problem solving tasks (Shneiderman, 1980). The interactive computer program debugger has the ability to: (a) supplement constrained short term memory resources, (b) provide increased learner control and feedback, and (c) provide visual images of the source program execution images. Some research evidence may suggest that some type of learners, field dependent students, may benefit from the interactive program debugger more than other students.

This chapter will review historically significant and current research related to this study. The first section of this chapter will review research into computer program debugging. The second section of this chapter will review current studies into problem solving and short term memory.

The third section of this chapter will introduce the concept of cognitive style and will be followed by a review of research literature into field dependence and programming. The fourth section will review current studies into learner control. The last section will summarize the review of literature as an orientation for the present study.

#### Computer Program Debugging

Philip Gilbert (1983) described debugging a program in following manner:

When the processing of a test point gives a different result from the specified one, an error has been found. That is, the test point has been successful in discovering that the program has a fault. The error must now be precisely pinpointed and repaired, a process called debugging. The error itself is called a bug. (p. 545)

Referring to the debugging process Roger Pressman (1987) stated that:

Results are assessed and a lack of correspondence between expected and actual is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction. (p. 519)

Pinpointing the location and the nature of a program bug (logic error) is the most time-consuming task in debugging. Meyers (1979) has said, "locating the error is 95 percent of the problem" (p. 257). Roger Pressman (1987) provided a list of reasons why program debugging is so difficult (see Table 7).

Table 7

**Psychological Difficulties in Program Debugging**

1. The symptom and the cause of the program bug may be geographically remote. That is the symptom may appear in one part of the program while the cause may actually be located at a different location within the program.
2. The symptom may disappear temporarily when another error is corrected.
3. The symptom may be caused by a non-error, i.e., round-off inaccuracies.
4. The symptom may be caused by a human error that is not easily traced.
5. The symptom may be a result of the indeterminate order for interactive data entry, and the error in the input conditions may be difficult to reproduce.
6. The symptom may be caused by hardware errors, or the interaction of hardware and software.

Note: Adapted from Roger Pressman (1987), Software Engineering: A practitioner's approach, New York: McGraw-Hill, pp. 521-520.

Describing program debugging from the psychological perspective Shneiderman (1980) states:

Debugging is one of the most frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors increase the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected. (p. 28)

Research into program debugging is a relatively recent development. Benander & Benander (1989) studied the

frequency of using of five traditional mainframe debugging techniques taught in education settings. Table 8 summarizes their findings. Studies of microcomputer program debugging strategies were not found.

<b>Table 8</b> <b>Traditional Program Debugging Techniques</b>	
Hand Tracing	The manual tracing of execution of a program and the contents of variables at various stages of the program's progress.
Appropriate Output Statements	The use of source language output statements to test the execution of critical points within a computer program. Source language output statements can also be used to provide an automatic trace of values at various stages of the programs.
Debug Verbs	The use of compiler or program translator options to help debug a program.
	STATE- Causes the COBOL statement that was being executed at point of the error to be printed.
	FLOW and TRACE- Causes a printing of the procedure names executed before the error.
	XREF - Causes a sorted listing of data names and procedure names to be printed.
Seek Help From others	Ask other students, tutors or faculty for help.
JCL Abend Codes	Studying the operating system's ABEND codes (abnormal termination) to determine why a particular program step did not execute.

**Note:** Adapted from Benander & Benander, 1989.



Using a questionnaire method, Benander & Benander (1989) found that students who utilize hand tracing techniques compared to other program debugging techniques were the most successful and required the least amount of program debugging time. Those students who reported the greatest amount of debugging time were inconsistent in the debugging method used and relied on the non-hand tracing techniques. The Benander & Benander study also found that students who utilized hand tracing techniques, more than any other program debugging techniques, resulted in the least amount of time for debugging and were the most successful. The assumption of the study was that it was the students' failure to understand their program logic influenced their choice of a debugging technique.

Doris Carver (1989), investigated the patterns of program debugging used by three professional programmers while coding and developing 13 COBOL modules. The study did not intend to produce any generalized conclusions. Rather, it proposed a new measurement instrument for future debugging research, called the Programmer Change Profile (PP). The PP coefficient can be used to describe the pattern of debugging corrections used by different programmers. For example, some programmers implement a high number of changes early in the debugging process and few changes later in the debugging process. Other programmers in the study exhibited a more constant rate of change over the entire debugging

cycle. Carver, proposed the possibility that there exists a psychological point, called the "Change Saturation Point," for every programmer where the programmer will not proceed with any more changes before submitting the program for further execution and testing.

Program debugging strategies represent a set of cognitive processes whose objectives are to locate a program error and to formulate a solution that will correct the error (Shneiderman, 1980). Program debugging strategies employ various disembedding and various cognitive reconstructing strategies. Meyers (1979) categorized three popular types of program debugging strategies: brute force, backtracking and cause elimination.

Roger Pressman (1987) described a "brute force" debugging strategy as:

Using a 'let the computer find the error' philosophy, memory dumps are taken, run time traces are invoked, a program is loaded with WRITE statements. We hope that the morass of information that is produced will find a clue that will lead us to the cause of the error. Although the mass of information produced by this method may eventually lead to the discovery and correction of the program bug, it frequently leads to wasted time and effort. (p. 521)

A memory dump is a diagnostic report that displays the exact binary contents of the internal computer's memory at the point of the error. These reports are useful for detecting many program execution errors, e.g., divide by zero (Davis, 1983).

The TRACE statement is a COBOL source command, which is coded in the program and will display the major execution steps (modules or paragraphs) while the program executes. The output of the TRACE statement is reviewed by the programmer to investigate the major processing steps executed by the program. The TRACE statement does not display detailed program code execution.

The WRITE statement is a COBOL source program commands that directs output to the printer. Another COBOL statement DISPLAY, which will direct output to the screen, is more frequently used in current COBOL text books. Stern & Stern (1981) recommends the use of the COBOL DISPLAY statement to check for logic errors.

To make debugging easier, it is possible to examine the contents of certain fields at certain checkpoints in the program, usually after the fields have been altered. In this way the programmer can easily spot a logic error by manually performing the necessary operations on the data and comparing the results with the computer-produced output that is displayed. When a discrepancy is found, the logic error must have occurred after the previous check point. (Stern & Stern, 1991, p. 383)

Pressman (1987) described "backtracking" as a common program debugging strategy that has been found to be generally successful in the debugging of smaller programs. Beginning at the point in the program where the symptom is encountered, the source code is manually traced back until the location of the program bug is encountered. Unfortunately, as the number of lines of program code and

the interaction of program bugs increases, the effectiveness of backtracking decreases.

The third category of program debugging proposed by Meyers (1979) is "cause elimination." Cause elimination program debugging strategies require the programmer to gather data and to prepare a list of possible causes for an observed program bug. Tests will be designed and conducted in an attempt to isolate the bug.

There are critics of interactive debugging tools. Swaine (1990) claims that debugging tools are overrated and are often abused by poor programmers. Knowledge of proper program structure and the ability to analyze data flows are the best ways to write a program and to detect program errors. There are additional concerns (Dijkstra et al., 1989) that the automation of computer science curriculum may create a situation where students may no longer have any concrete understanding of the actual processes of writing programs.

#### Problem Solving and Short Term Memory

Shneiderman (1980) has described program debugging as a problem solving task. Problem solving tasks have been categorized by the procedures used to achieve a solution based upon a particular set of problem requirements (Bourne et al., 1986). Within this context, writing computer programs is a transformation process, which may utilize

various programming algorithms to solve a particular program requirement.

Previous research has not provided evidence to support the existence of any one best problem solving strategy. Simple problems may be solved by applying various heuristic problem solving strategies, e.g., representiveness (Ormrod, 1990). This short cut method compares the similarities of current program requirements to previously written programs or algorithms. Complex problems frequently require a combination of algorithmic procedures (Ormrod, 1990).

Breaking up a complex program into two or more subproblems and then working successively on each individual component is an illustration of top down problem solving. Top down program design strategies (Pressman, 1987) are applications of "means-ends" analysis (Newell & Simon, 1972; Restle & Davis, 1962; Resnick & Glasser, 1976). "Divide and conquer" cognitive strategies seek to work within the constraints of short term memory resources.

Previous cognitive research has shown that short term memory capacity is a bottleneck for any problem solving process (Ormrod, 1990). If the amount of internally stored information and the cognitive requirements of problem solving strategies exceed the capacity of short term memory, the problem cannot be solved (Ormrod, 1990).

Short term memory has a very limited storage

capacity. The maximum number of information units that can be stored is approximately seven, plus or minus two (Miller, 1956). Simon (1974) suggested that chunking, a process of combining information units, can be used to effectively increase the short term memory storage. Language based information is generally stored in a more efficient, or compressed, auditory or verbal format (Conrad, 1971). Some short term information may be stored in visual form (Conrad, 1972). Without rehearsal, short term memory will retain information for about 20 to 25 seconds (Peterson & Peterson, 1959).

#### Cognitive Style, Field Dependence and Programming

This section begins with a discussion of cognitive styles and the nature of field dependence. This is followed by a review of the research in field dependence and programming, and is concluded with a discussion of the measurements used for field dependence.

Not all individuals learn in the same manner. Students tend to persistently use a learning strategy that best fits their cognitive needs to acquire knowledge. Learning is an individualistic cognitive activity. In 1971, Witkin, Oltman, Raskin and Karp define cognitive styles as:

. . . the characteristic, self-consistent modes of functioning which individuals show their perceptual and intellectual activities. These cognitive styles are manifestations in the cognitive sphere of still broader dimensions of personal functioning which cut across diverse psychological areas (p. 127).

Messick (1954) stated that cognitive styles:

. . . represent consistencies in the manner or form of cognition or the level of skill displayed in cognitive performance. They are conceptualized as stable attitudes, preferences, or habitual strategies determining a person's mode of perceiving, remembering, thinking and problem solving. (p. 5).

While researchers have suggested numerous constructs to describe the differences in student's cognitive styles, field dependence seems to be the most appropriate for the study of interactive program debuggers.

Messick (1954) described field independent individuals as being capable of solving problems that require them to take a fact out of context and then restructure the information to be used in a different context. These cognitive abilities, used by field independent individuals, are frequently called "disembedding" and "cognitive restructuring" (Cavaiani, 1989, p. 412). A field independent individual addresses the environment in more analytical terms and can more easily find the presence of logic errors (Messick, 1954). Field independent individuals also enjoy working things out themselves, prefer a solitary environment and require less feedback. During problem solving processes these individuals will perceptually and intellectually analyze and impose structure on an unstructured task.

On the other hand, field dependent individuals have trouble disembedding objects from their context, perceive constructs more globally and rely on external references

(Messick, 1954, Rameriez & Castaneda, 1979). During problem solving processes these individuals accept an unstructured task as they perceive it and have difficulty analyzing and structuring the task (Witkin, Goodenough & Oltman, 1979). Pascual-Leone et al. (1978) have shown that field dependent individuals select inappropriate problem solving strategies more than field independent individuals irrespective of the situation.

Cognitive styles are not value directional. Field independent individuals are not considered to have superior abilities than field dependent individuals, rather these individuals process and perceive information differently. Witkin & Goodenough (1981) noted that the nature of field dependence is not easily altered and that it remained stable over a period of years. Jonassen (1987) recommended that instructors provide a variety of learning activities to allow students to encode and interpret information in a way that best takes advantages of one's particular cognitive style.

Jonassen (1988) and Cronbach and Snow (1978) have suggested that the learning environment should be adapted or supplanted to take into account the strengths and weaknesses of the student in order to achieve a desired instructional outcome. Supplantation approaches prescribe that the instructional process would be more effective when the tasks



and methods of presenting information are designed to complement the internal processing skills of the student.

French (1983) has identified two types of supplantation processes: conciliatory and compensatory. Conciliatory approaches are designed to emphasize the strengths and to avoid the weaknesses of the learner. For example, the utility of the interactive program debugger to visually display the student's program code while it was executing was expected to assist field dependent students in learning program debugging skills.

On the other hand, compensatory supplantation requires the instructional designer to provide instructional tools required for a task that the learner does not possess. For example, the utility of interactive program debugger to: (a) inquire and manipulate multiple variables, (b) highlight, cue and set interactive breaks points in an executing program, and (c) the animation of a program structure chart was expected to assist field dependent students in learning program debugging skills.

Cavaiani (1989) investigated the influence of field dependence on the ability of a student programmer to debug a COBOL program. Thirty-nine students enrolled in an introductory COBOL programming course using a mainframe computer participated in this study. Using the Group Embedded Figures Test, Cavaiani (1989) assigned the participants to one of two groups, field independent and

field dependent. Seven debugging programs were administered as a pen-and-pencil test to the students on several different occasions during the semester. Two of the test programs contained a syntax error. Five of the test programs contained a program logic error.

The debugging test score was based upon a weighting of three different criteria: (a) the number of errors located and corrected, (b) the number of errors located and not corrected, and (c) the number of correct statements that were marked incorrect. Separate composite scores were compiled for the syntax debugging and the logic debugging tasks.

Using a Spearman Correlation Analysis, the Cavaiani study (1989) provided evidence that there was no difference between a field independent and a field dependent individual in the ability to correct syntax errors in a computer program. This result was expected since this type of programming debugging task required minimal effort and the task could be completed in the given contextual environment. On the other hand, field dependent individuals did have significantly more difficulty in locating and correcting program logic errors.

The Cavaiani study (1989) also investigated the effects of various weighting schemes to be used for scoring program debugging tasks. In order to differentiate between field independent and field dependent programmers, Cavaiani

suggested that the program debugging scoring scheme possess the following characteristics: (a) it should assign the same or nearly same weight to the location and correction of the error as it does to finding the location of the errors only, and (b) it should assess a penalty to subjects who make use of trial-and-error methods. Weighting schemes used in the study that exhibited the above criteria were able to statistically ( $p < .05$ ) detect a difference between field dependent and independent programmers in terms of the ability to locate and correct a program logic error.

Identifying syntax errors is a trivial task and relies more on memory recall than a problem solving process. However, locating and correcting program logic errors requires the individual to take a critical program element out of the context of the program and to select the correct problem solving strategy to formulate a solution for the logic error. This process also requires students to understand the logic error in relation to the context of the program.

Measuring the construct of field dependence can be categorized into 2 types: 1) perceptual tests such as the Body Adjustment Test and the Rod and Frame Test, and 2) general fluid visualization tests such as the Embedded Figures Test and the Group Embedded Figures Test (Linn & Kyllonen, 1981).

In early research into field dependence, Witkin (1949) studied the ability of pilots to keep their orientation in relationship to the ground. These original tests were conducted under specialized testing conditions. The Body Adjustment Test (BAT) required an individual who was seated in a small room to adjust his body's posture to a true upright position when both his chair and the room were tilted in different directions. The Rod and Frame Test required an individual who was seated in a dark room to direct the experimenter to adjust the position of a lighted rod until it was vertical. A portable version of the Rod and Frame Test was subsequently developed to aid researchers in the field.

Witkin et al. (1971) developed two alternative testing procedures that could be completed with pencil and paper. The Embedded Figures Test (EFT) and the Group Embedded Figures Test (GEFT) required a subject to locate a simple figure within the context of complex and obscured field. The EFT was individually administered to each subject and required them to trace the sought-after simple figure on the test card. The EFT was impractical to administer to a large number of subjects as required in large scale research. A large group version of the EFT was subsequently developed.

The Group Embedded Figures Test (GEFT) was modelled as closely as possible to the Embedded Figures Test (EFT) with respect to presentation and format. Light shading of areas

of the figures replaced the use of color to obscure figures in the EFT. Results have shown that men perform slightly, but significantly, better on the GEFT test than women ( $p < .005$ ). Results of the test tend to be more reliable over the age of 17. These findings are consistent with those differences found with the EFT (Witkin, et al., 1971).

The Group Embedded Figures Test (GEFT) correlates highly with the Embedded Figures Test (-.82 for men and -.63 for women). The GEFT also significantly correlates with the Portable Rod and Frame Test (-.39 for men and -.34 for women).

### Learner Control

Historically, the amount of control that a student programmer can exercise with the traditional program debugging methods has been limited. The interactive program debugger may permit a student to control the debugging process in a manner that best meets their particular learning style. The concept of learner control has been a topic of considerable research interest in computer-based instruction. Previous research indicates that learner control will provide benefits in terms of motivation, interest, enhancement of metacognitive and cognitive skills, and adaptation to particular learning style preferences (Carrier & Jonassen, 1988; Jonassen & Tennyson, 1983; Lee, 1991).

Research results concerning learner control and achievement are more complex. Previous studies have indicated that increased learner control had no effect on achievement when college students had a low level of prerequisite knowledge. On the other hand, high levels of prior knowledge and learner control treatments, were found to increase achievement in college students (Goetzfried & Hannafin, 1985, Hannafin, 1984, Krendel & Liberman, 1988, Steinberg, 1977, Tobias, 1976).

Considering the interaction of metacognitive factors and learner control, Lee (1991) provided evidence that a child's prior knowledge may not be a factor in the acquisition of information. Learner control strategies when supplemented by appropriate feedback and supportive factors, e.g., clearly labeled options and advice of ongoing progress, can be effective even for novice learners. In a related study, Arnone & Grabowski (1991) presented evidence citing that learner control strategies with "advisement pre-lessons" will provided the greatest level of achievement and curiosity in younger children.

Matton et al. (1991) noted that previous findings on learner control strategies failed to consider the nature of the instructional objective. Using a flight simulator application designed for the United States Air Force, Matton et al. (1991) found no positive learner control effects. The results of this study suggested that problem solving

learning may not benefit from learner control as much as concept and rule based learning.

### Orientation to the Present Study

Previous research on student and professional program debugging skills has been too inadequate to generalize any conclusions related to factors that affect the ability of a student to locate and correct a logic error (debug) in a computer program. In addition, these prior studies have failed to consider newer techniques of debugging programs, i.e., the interactive program debugger.

More specific research is needed into program debugging. Previous researchers (Benander & Benander, 1989; Cavaiani, 1989) used students who may have had prior programming experience in other courses. These studies, however, had failed to consider the degree to which programming experience affected the choice of the debugging tool or whether programming experience affects the proficiency of the use of the debugging tool. In addition, the Cavaiani study (1989) used a pen-and-pencil test to measure the student's ability to locate and correct a program logic error. This artificiality of the testing condition may limit one's ability to generalize the results of this study to the actual program debugging task performed on a computer.

Research is also needed into the trend toward microcomputer-based programming education. The Benander &

Benander's study (1989) was focused on the use of traditional mainframe computer programming tools. This study will investigate both traditional and interactive program debugging tools that are available on microcomputers. The low cost of microcomputer hardware and programming software and their ease of use has made microcomputers very popular for many programming language courses. Though mainframe computers support interactive program debuggers, they generally do not support the graphical type interfaces used by microcomputer program debuggers.

Successful program debugging begins with the programmer's ability to perceive the symptoms of the program logic error. The utility of the interactive program debugger to visually animate the execution of the program code may help student's to better perceive and locate the logic error.

The interactive program debugger also may lessen the cognitive burden on short term memory resources of student programmers. Relieved of various clerical tasks, student programmers may be able to concentrate more on the process of learning program debugging. Interactive program debuggers lessen cognitive burden by: (a) enabling the student control the speed of program execution by pressing a key or to setting the speed of execution and (b) automatically monitoring various data entity changes while the program executes.



Previous research (Cavaiani, 1989) seems to suggest that some student programmers may benefit more than others. Field dependent individuals should be expected to benefit more from the use of interactive program debuggers than field independent individuals. Field dependent students were expected to be aided by several possible disembedding and constructive strategies employed by the interactive program debugger: (a) the ability to view the execution steps of the program as written in the student's native program code, (b) the use of various organizational cues that would enable students to structure the debugging process, e.g., color highlighting of an executing statement organizes attention, and (c) inspected data items are graphically boxed to disembed them from the context of the executing program (Appendix J).

Previous research has provided evidence for the instructional advantages of increased learner control (Goetzfried & Hannafin, 1985, Hannafin, 1984, Krendel & Liberman, 1988, Steinberg, 1977, Tobias, 1976). The ability to control variables is the foundation of the experimental learning process. Other sciences have conducted real time experiments that have enabled individuals to learn concepts and skills by manipulating various variables during the experimental process. In the past, similar real time experimentation strategies were not available to computer programmers. Increased student control of the debugging

process may encourage the development of an organized program debugging strategy, a "cognitive map," that may be successfully retrieved in the future.

In this study, students will be exposed to two different instructional methods, interactive and traditional program debugging, used in two different programming language curricula (COBOL and BASIC). Students will further be classified as field independent, indeterminate, or field dependent. This study will attempt to see if there exists a difference between the interactive and the traditional instructional methods, and whether the type of programming language and the level of field independence interacts with the student's ability to locate and correct logic errors in a computer program.

## RESEARCH METHODOLOGY

### Problem Restated

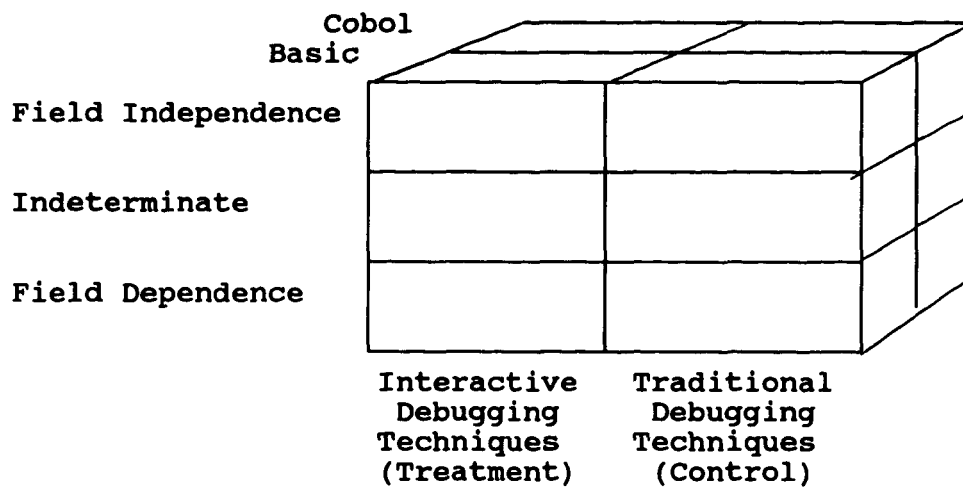
The purpose of this study was to evaluate the effects of computer program debugging tools, computer program languages, and field dependence on the ability of a student programmer to locate and correct logic errors in a computer program.

### Research design

The design of this study is a posttest-only, completely between-subjects, fully-crossed factorial involving three independent variables: (a) debugging treatment, (b) programming language, and (c) field dependence. A quasi-experimental strategy was used to study the effects of these independent variables on two dependent variables: (a) ability to locate and correct a program logic error (LOCCOR) and (b) the amount of time to successfully locate and correct a program logic error (TIMECOR).

FIGURE 1

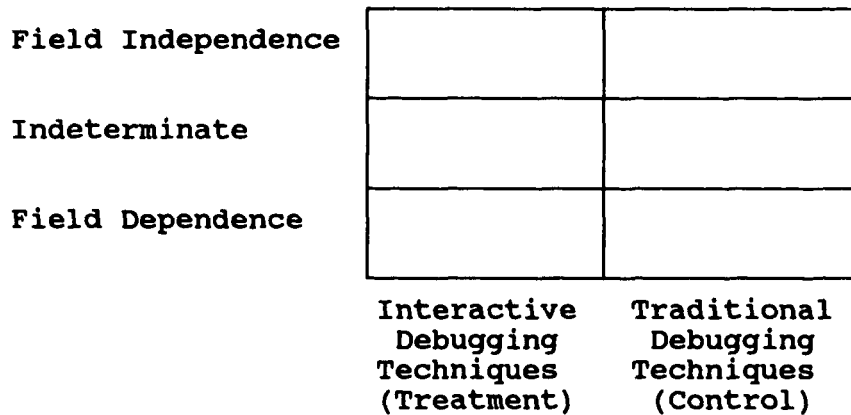
**A 2 X 3 X 2 Factorial Analysis of the Ability to Locate a Computer Program Logic Error**



[Covariate: Prerequisite Programming Skills]

FIGURE 2

**A 2 X 3 Factorial Analysis of the Time to Locate a Logic Error in a COBOL Program**



[Covariate: Prerequisite Programming Skills]

FIGURE 3

**A 2 X 3 Factorial Analysis of the Time  
to Locate a Logic Error in a BASIC Program**

Field Independence		
Indeterminate		
Field Dependence		
	Interactive Debugging Techniques (Treatment)	Traditional Debugging Techniques (Control)

[Covariate: Prerequisite Programming Skills]

A factorial design permits the analysis of complex behaviors. Based upon Issac & Michael's guidelines (1981), a factorial research design was appropriate for this study because: (a) the simultaneous interaction between independent variables (e.g., treatment and field dependence) may affect the response variables, (b) field dependence could not be controlled in the design of the study (i.e., it cannot be directly manipulated), and (c) several research questions may be tested simultaneously.

### Methodological Limitations

A quasi-experimental research design attempts to approximate the conditions of a true experiment, but is conducted in a setting that does not allow the researcher to control or manipulate all relevant variables. While quasi-experimental research designs may closely approximate the experiences of a "real world" education system, this research design may pose threats to the internal validity of the study (Maxwell & Delaney, 1990). Among the conditions limiting our ability to measure the effects of the treatment on the dependent variables include: (a) possible selection biases attributable to the use of intact groups, (b) the mastery of programming language prerequisite skills necessary to debug computer programs, (c) controlling student activities outside the classroom, (d) controlling the use of the interactive program debugger in the BASIC programming language treatment, and (e) the threats to internal validity caused by a posttest-only research design. Each of these limitations is discussed below.

Due to the limitations of selecting the subjects in a college level academic higher education environment and the limited availability of computer facilities, the random assignment of subjects to groups was impossible. Students registered for the various course sections in a normal fashion and without manipulation. The COBOL sections used in

the study were randomly selected from the pool of all available sections. The BASIC sections used in the study were selected at random from a pool that represented 70% of all available sections. Once the sections were selected, each intact section of COBOL and BASIC programming classes were randomly selected and assigned to either the interactive or traditional debugging treatments. No student was enrolled in more than one section.

A consequence of using intact groups is that there is little control over the characteristics of the subjects selected. The variable PROGEXP was expected to correlate with each dependent variable, and represented the number of computer programming courses at the high school, collegiate, or professional level taken by the participant prior to the beginning of the study. Because this variable could conceivably contaminate interpretations of the data, it was important to try to control its effects statistically by using PROGEXP as a covariate in an ANCOVA. Other background and demographic data (Appendix F) also were collected by a pre-course survey and were analyzed to determine if they should be used as covariates in the analysis.

A second shortcoming of the design used in this study involves the participant's mastery of minimum program development, syntactical and algorithmic skills. To take into account the participants previous mastery of programming skills, three program language prerequisite

knowledge tests were administered prior to the treatment and were subsequently analyzed to determine their correlation with the dependent variables used in this study. An ANCOVA was used to adjust the difference in means of the dependent variables attributable to the differences found in the mastery of computer programming prerequisite skills. These COBOL and BASIC computer programming prerequisite skills are outlined in Appendices A and B.

A third limitation of the research design was the lack of ability to control student activities outside the classroom. Conducting computer educational research over a period of time in which students may use the computer outside the classroom posed other threats to the internal validity of this study. These threats include: (a) controlling communications between students involved in the treatment and control groups and (b) controlling the amount, type and quality of time spent outside the class preparing, coding and debugging programs. To assess student use of the computer outside the classroom student activity data were collected during the study, which required students to report the amount of time that they spent studying and programming outside the class lectures. Students were required to fill out a study and programming activities journal (Appendix G) in class on a weekly basis. Preparing the activity journal during class time room offered the advantage of providing consistency in instruction between



subjects. To minimize student manipulation of the journal data directions were provided with the activity journal, which informed students that the activity data collected from the journal would not influence their grade.

These weekly study and programming activity journal data were used to measure the amount of time spent studying, programming or debugging their assignments. In addition, students were required to report any communication or assistance received from other students, tutors or faculty members. This evidence was reviewed to determine the extent to which communication with individuals outside the scope of the study, and between the various treatment groups during the debugging treatment may have posed a threat to validity of the study.

Note that the reliability and validity of the programming activities journal data gathered could be influenced by: (a) the student's ability to accurately recall the previous week of study and programming activities and (b) the student's desire to distort the data to influence their grade. External verification of the activity journal data was limited.

A fourth limitation of the study design was the lack of ability to control the student use of the interactive program debugger in the BASIC programming courses. The MicroSoft QuickBASIC course text book provided a complete, student version of the QuickBASIC programming language,

which included an interactive debugger option. A faculty survey on the use of interactive program debuggers (Lavery, 1990) showed that an interactive debugger had not been used in previous BASIC programming courses at the University of Pittsburgh. Data from the activity journals provided evidence that actual student use of the QuickBASIC interactive debugger outside the scope of the study was insignificant.

Limiting student access to the COBOL interactive program debugger presented minimal control problems. Access to the COBOL interactive program debugger was only available from the Robert Morris College's network and was restricted by an individual student's password.

The fifth limitation of this research design was the threats to internal validity posed by a posttest-only research design. A posttest-only research design fails to control for selection, history and maturation of the subjects (Issac, & Michael, 1981). However, the threats to internal validity of a posttest-only design were considered to be less than the testing effects of the alternative pretest design. It was expected that the more a student practices debugging a program the more proficient they become.

### Subjects

Students in two sections of CI201 Business Programming, offered by the Computer Information Systems Department of Robert Morris College, participated in the COBOL programming language section of this study (see Table 9). This group of students represented approximately one half of the students enrolled in the CI201 Business Programming course taught at the Moon Township campus in the Winter Term of 1992 (n=40). Students in two sections of CS4/007 BASIC Programming, offered by the Computer Science Department of the University of Pittsburgh, participated in the BASIC programming language section of this study (see Table 9). This group of students represented approximately 15 percent of the students enrolled in CS4/007 BASIC Programming course taught at University of Pittsburgh's main campus (n=45). Only students meeting minimal attendance standards during the semester and the treatment period were used in the study. The same instructor conducted all four course sections in the study.

All sections were offered during the daytime, and no student was enrolled in more than one section. Based upon prior registration experience the demographic and background characteristics of the Robert Morris College and the University of Pittsburgh students were expected to be comparable.

Table 9	
Alternative Instructional Formats	
Instructional Section	Description
1	COBOL Programming Traditional (SECT1) [Robert Morris College, n=20]
2	COBOL Programming Interactive (SECT2) [Robert Morris College, n=20]
3	BASIC Programming Traditional (SECT3) [University of Pittsburgh, n=23]
4	BASIC Programming Interactive (SECT4) [University of Pittsburgh, n=22]

The CI201 Business Programming course is described in the Robert Morris College course catalog as "an introduction to structured COBOL and programming techniques. Logical structure, modular design and documentation techniques are presented. The student becomes familiar with the syntax and logic of COBOL by applying the language to a sequence of increasingly complex business applications." A computer literacy course is a prerequisite for this class. The COBOL participants in this study used SPFPC, a microcomputer editor, to write their COBOL programs. The COBOL participants were then required to translate their COBOL source programs into executable code using MicroFocus COBOL, a microcomputer ANSI 85 program checker, as installed on

Robert Morris College's Novell network. Students were unable to translate or execute their COBOL programs at home.

The CS4/007 Basic Programming course is described in the University of Pittsburgh's course catalog as "the first course in computer science. It is designed to be of special interest to students majoring in one of the social sciences or humanities." No prerequisite computer science courses were required for this course. The BASIC programming participants prepared and executed their BASIC source programs using MicroSoft QuickBASIC's editor and interpreter. The course text included a student version of MicroSoft's QuickBASIC, which students could have used at home. The BASIC programming students also could have used the University of Pittsburgh's Novell network located in the computer laboratories to prepare and execute QuickBASIC programs.

Initial COBOL and BASIC class size was 28 and 35 students, respectively. Classroom attendance data and program assignment grades were collected and monitored. Minimum classroom attendance and program assignment performance were necessary to determine the effects of the debugging treatment. Students below minimum attendance and assignment standards were not be used in the study. After adjusting for mortality, the size for each of the four language treatment/language groups ranged from 20 to 23 students.

The experienced mortality rate for each section was not greater than the normal mortality rate found in previous BASIC and COBOL courses taught at the University of Pittsburgh and Robert Morris College. However, the level of field independence and prior programming experience were greater for those students who remained in the study than those students who had either dropped the course or who were rejected from the study for not meeting minimal selection criteria. While presenting the debugging treatment earlier in the semester may have decreased the rate of mortality, the timing of the debugging treatment was not addressed in this study.

It is important to determine the appropriate sample size adequate to compare the effects of the interactive and traditional debugging treatments and to specify how large of a difference was "statistically" and "practically" acceptable (Levin, 1975). Considering the nature of a quasi-experimental research design, the moderate sample size and opportunity cost of the additional instructional time necessary to present the interactive program debugger, a reasonably large treatment effect would have been necessary to be of practical significance. Since the sample size available for the study was fixed, sample size and power calculations were, necessarily, of a post hoc nature. Using Cohen's (1969) guidelines, the .38 standard deviations difference which was found between the interactive and the

traditional debugging posttest mean score (LOCCOR) would be considered a "medium" effect size.

For most experiments the Pearson and Hartley power charts (1951) can be used to determine the statistical power of a test for a specified sample size, observed effect size and level of significance. The power of a test is the "probability of rejecting the null hypothesis, when the alternative hypothesis is true" (Kirk, 1968, p.3). The statistical power of a test is equal to one minus the probability of a Type II error (Kirk, 1968).

Since the prime interest of this study was to study the effects of interactive and traditional debugging tools, the statistical power of this contrast was important. The size of the sample used in this study to compare the use of the program debugging tool was approximately 40 participants. The Pearson and Hartley power charts (1951) were used to determine the level of statistical power provided by a test of a fixed sample size of 40. To use the Pearson and Hartley power charts (1951) the level of significance, the degrees of freedom and a noncentrality parameter,  $\phi$ , must be specified. In this study, a .05 level of significance was used to test the statistical hypotheses.

Since an accurate estimate of the population variances was not available from previous research, an alternative formula for  $\phi$ , developed by Kirk (1968) was used. Given the observed .38 standard deviation units found between the

debugging test scores, the value of phi was calculated to be 1.69. Using this phi value and the Pearson & Hartley power chart, the statistical power of the F Test was estimated to be .86. Given an effect size of .38 standard deviations and a statistical power of .86, it was estimated that there was an .86 probability of rejecting the null hypothesis when the null hypothesis is in fact false. In other words, if a program debugging tool treatment effect did exist, the F Test had an 86% chance of detecting this effect.

### Instructional Materials

The design of the CI201 Business Programming course outline, syllabus and prerequisite materials was based upon the Robert Morris CIS Departmental syllabus and course text. The CI201 Business Programming course outline, syllabus (Appendix D) and prerequisite materials that were used in the study are those that were currently being used by faculty members of the Robert Morris CIS Department.

The design of the Computer Science BASIC Programming course outline, syllabus and prerequisite materials was based upon the University of Pittsburgh Computer Science Departmental syllabus and course text. The CS4 BASIC Programming course outline, syllabus (Appendix E) and prerequisite materials that were used in the study are those



that were currently being used by faculty members of the University of Pittsburgh Computer Science Department.

Faculty members from the Robert Morris College Computer Information Systems department and the University of Pittsburgh Computer Science department were be asked to review the course outline, schedule and instructional material to determine if: (a) The course content was appropriate for the level of debugging skills tested, (b) the sequence of the course material was appropriate to teach programming debugging skills to be tested, and (c) whether there was adequate instructional time assigned to the program debugging instructional task that would permit students to learn debugging techniques.

Based upon the recommendation of interviewed faculty members, the amount of instructional time allocated to the concepts of program debugging was slightly increased. Faculty members from the Robert Morris College Computer Information Systems department and the University of Pittsburgh Computer Science department also were asked to review the modified computer debugging instructional materials for each language-treatment group. These faculty members did concur that these materials ensured a comprehensive instructional program. At the recommendation of the faculty members at both schools, the timing of the debugging instructional treatment was postponed from the

tenth week of the semester to the thirteenth week of the semester.

Separate pilot groups of COBOL and BASIC programming students were presented with the modified instructional materials in a classroom environment, subsequently interviewed, and asked to make recommendations that would have improved the quality of the program debugging instructional materials. No significant changes to the debugging instructional materials were necessary.

### Procedures

Table 10 outlines the procedures used to administered the tests and treatments during the study.

Table 10

**Schedule of Surveys and Tests  
Used in the Study**

Week 3	Pre-Course Student Survey	Appendix F
Weeks 3 thru 14	Student Activity/Programming Journal	Appendix G
Week 7	Prerequisite Test One	
Week 10	<u>Group Embedded Figures Test</u>	
Week 11	Prerequisite Test Two	
Week 13	Prerequisite Test Three	
Week 13	Debugging Treatment	
Week 15	Post-Experiment Program Debugging Test	Appendices H&I

Before the semester began, the proposal for this project was approved by the departmental chairperson of the Computer Information Systems department at Robert Morris College and the departmental chairperson of the Computer Science Department at the University of Pittsburgh.

A Student Background Survey (Appendix F) was administered to the students during the third week of the course.

Students were required to fill out a study and programming activities journal (Appendix G) in class on a weekly basis. The journal data were collected from the third week until the fourteenth week of the semester.

The Group Embedded Figures Test (Witkin, et al., 1971) was administered during the tenth week of the semester. The three program language prerequisite tests were administered to students during the seventh, eleventh and thirteenth weeks of the semester. Students were required to take any make-up tests within one week of the original test date. Test results were not returned to students until all make-up tests had been administered.

During the thirteenth and fourteenth week of the semester students received lecture, instructional materials and in-class program debugging exercises designed for each debugging treatment and programming language. An out-of-class program debugging exercise was assigned to students to provide hands-on computer programming debugging skills practice for each debugging treatment. This out-of-class program debugging exercise was graded by the instructor and constituted five percent of their final grade for the course. Students not completing this computer debugging skills assignment satisfactorily (an 80% grade) were not used in the study since it was considered that they were not adequately exposed to full debugging treatments from which valid conclusions could have been reached.

After the debugging treatment, a program debugging posttest (Appendices H & I) was administered to the students in the fourteenth week of the semester. This test was

conducted during normal class time in a computer equipped classroom. All test work was performed on the computer.

### Instruments

#### Pre-course survey

A Pre-Course Background Survey (Appendix F) was administered to all students to gather basic demographic and computer background information, including prior programming experience.

#### Studying and programming activity journal

Students were required to fill out a study and programming activities journal (Appendix G) in class on a weekly basis. Data gathered from this journal were used to measure the amount of time students spent debugging programs during the debugging treatment and also provided evidence to help answer research question two.

#### Program language prerequisite test

The ability to prepare a program using the required syntax rules for a given programming language is an essential prerequisite skill to debug program logic errors. Data gathered from the program language prerequisite tests were used to measure the student's mastery of prerequisite programming skills.

Standardized test items designed for the specific versions of MicroFocus's COBOL and MicroSoft's QuickBASIC were not available. Objective test questions were selected from previously administered tests used in other CI201 Business Programming and CS4 BASIC program courses. The program language prerequisite tests were reviewed by Robert Morris College CIS and University of Pittsburgh Computer Science faculty members to assess content validity. No changes to the prerequisite tests were recommended.

Three program language prerequisites tests were administered to students in each section of CI201 Business Programming on three different test dates. Three different program language tests were administered to students in each section of CS4 BASIC on three different test dates.

The objective of the first program language prerequisite test was to measure the basic program development process and elementary syntax rules for each respective programming language. The objectives of the second and the third tests were to measure advanced syntax rules and fundamental programming algorithmic skills, e.g., accumulation, high-low, etc. Except for the adjustments for differences in syntax, the test objectives were the same for both programming languages.

The program language prerequisite tests also were administered to two pilot groups of COBOL and BASIC

programming students prior to being administered to the participants. The results of the pilot test were analyzed by the Test Analysis program provided by the University of Pittsburgh's Office of Measurement and Evaluation. The pilot test items were reviewed to determine if: (a) the difficulty index was greater than ninety percent or less than ten percent or (b) the point biserial discrimination coefficient was less than zero. Based upon the results of this pilot test, seven questions were rewritten. Three other test items were retained in spite of the poor statistical performance since each question tested an important prerequisite skill. Data collected from the pilot language prerequisite tests were not used in the study.

Students were assigned a letter grade for their performance on the program language prerequisite test. This grade constituted 20% of their final grade for the course.

#### Program debugging test

Two program debugging posttests (Appendices H & I) were developed for each computer language. The same test was administered to both the interactive and the traditional sections in each respective programming language. This test was designed to measure: (a) the ability to locate and correct a program logic error (LOCCOR) and (b) the time to locate and correct a program logic error (TIMECOR). Data

gathered from the program debugging test assisted in answering all of the research questions.

The first step used in the construction of computer program debugging test was to interview faculty members of Robert Morris College CIS department and University of Pittsburgh Computer Science department to document the frequency and severity of program logic errors encountered by student programmers in an entry level computer programming course. Course texts also were reviewed to provide information concerning the nature and frequency of logic errors encountered by student programmers. Data were ranked ordered by frequency and were used to select program debugging tasks to be measured by the program debugging test.

Two pilot program debugging posttests were reviewed by Robert Morris College CIS and University of Pittsburgh Computer Science faculty members. These faculty members also determined whether the program documentation was adequate, clear and appropriate to complete the programming and debugging task for each program on the test.

The program debugging posttests were administered to a pilot group of CI201 Business Programming and CS4 BASIC students. Based upon the results of the pilot debugging test, the maximum time allocated to each test question was increased from twenty minutes to forty minutes. Data



collected from the pilot program debugging test were not used in the study.

The program examples and test examples were identical for all treatment groups, except adjustments made for the programming syntax requirements of each respective programming language. Logic errors caused by incorrect program syntax usage, e.g., misplaced period in a COBOL IF statement block, were not used.

Each debugging posttest consisted of five programs. The first program contained one simple logic error, e.g., failure to execute a statement within a loop. The second and third programs contained a more difficult logic error, e.g., incorrect sequence of statements. The fourth and the fifth programs contained two logic errors that interacted.

The computer debugging posttest was administered in a computer-equipped classroom. At the beginning of the debugging test, the test administrator provided each student with a floppy disk, containing the program source code and the input data file.

Each test program was be accompanied by appropriate program documentation. Using the guidelines developed by Nickerson (1986), the documentation provided to each student included:

- 1) a printed copy of the source program,
- 2) a printed copy of the input data file,
- 3) the description of the program requirements,

- 4) the description of the effects of the program logic error (cued recall format),
- 5) the current, incorrect printed outputs of the program, and
- 6) the required, correct printed outputs of the program.

All debugging test activities were performed on the computer.

Using the Pierson and Horn study (1984) as a guideline, the source programs used in the debugging post test were reviewed to ensure that no syntax errors were present. Each program was compiled (checked) and executed to ensure that only logic errors remained. None of the programs contained any syntax or execution errors. All test programs executed, but produced incorrect results.

Each section of the test was limited to forty (40) minutes for each debugging requirement before the student was required to continue to the next test program. At the end of each program section of the test, each student was required to fill out an answer-journal sheet. The student was required to describe the logic error found and to identify the debugging tools and methodology that they used to locate and correct the logic error. Students also were required to report the value of the debugging tool used on a rating scale that ranged from: one (1) not used, two (2) slight value, to five (5) very valuable.

Students were permitted to write on any of the printed documentation. Each student's program disk was collected after the test had been completed and was graded by the instructor to determine if the logic error had been successfully located and corrected. Each program was subsequently executed by the instructor to determine if the required program debugging tasks had been successfully completed. The operating system date/time stamp, stored with the program file name when the program file was last changed, was used to measure the amount of time spent to successfully locate and correct a program logic error (TIMECOR). The operating system date/time stamp also was used to ensure that students did not work on programs different from the assigned test sequence. Students who did not successfully locate and correct a program logic error were assessed forty (40) minutes, the maximum test time per debugging program.

Scoring schemes for the program debugging test were based upon the ones used by Cavaiani (1979) in his study of debugging program syntax and logic errors. A weighting system representing three different scoring criteria was used: (a) Failed to locate the error, (b) located the error and failed to correct the error, and (c) located the error and corrected the error. The weights for each test program increased as the complexity of the debugging task increased (see Table 11).

Program	Weight	Failed to Locate	Located Error	Correct Error	Max. Points
1	1	0	3	5	8
2	3	0	3	5	8
3	3	0	3	5	24
4	5	0	3	5	40
5	5	0	3	5	40

NOTE: Adapted from Benander & Benander (1989).

Students were assigned a letter grade based upon their performance on the debugging test. This grade constituted 15% of their final grade for the course.

All programming students and professional programmers need to be able to locate and correct a logic error within a computer program. The program debugging tests attempted to duplicate this cognitive task. The debugging test programs and documentation represented the kind of conditions that typically would be encountered by a student or professional programmer.

#### Data Sources

The data for this study was obtained from a pre-course survey, student activity journals, prerequisite tests, the Group Embedded Figures Test, the Debugging test score and

time. The variables used in this study are listed in Table 12.

Table 12	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>LANGUAGE</b>	Programming language taught in class section: COBOL coded 0, BASIC coded 1.
<b>TREATMENT</b>	Type of computer debugging treatment used in class section: Interactive program debugger coded 0, and Traditional program debugging coded 1.
<b>SECT1</b>	Students enrolled COBOL section receiving the traditional debugging treatment.
<b>SECT2</b>	Students enrolled COBOL section receiving the interactive debugging treatment.
<b>SECT3</b>	Students enrolled BASIC section receiving the traditional debugging treatment.
<b>SECT4</b>	Students enrolled BASIC section receiving the interactive debugging treatment.

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>AGE</b>	Current age of the participant, coded in years.
<b>SEX</b>	Gender: male coded 0, female coded 1.
<b>CREDITS</b>	Total number of college credits earned before the current course.
<b>MAJOR</b>	Major field of study: Undeclared coded 0, Social Science coded 1, Business coded 2, Math coded 3, Engineering coded 4, and Computer Science coded 5, Natural Sciences coded 6, Other coded 7.
<b>MINOR</b>	Major field of study: Undeclared coded 0, Social Science coded 1, Business coded 2, Math coded 3, Engineering coded 4, and Computer Science coded 5, Natural Sciences coded 6, Other coded 7.

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>BASIC</b>	Number of high school or college BASIC programming courses previously taken by student prior to course: Coded zero for none, Coded 1 for one course, coded 2 for two courses, etc.
<b>COBOL</b>	Number of high school or college COBOL programming courses previously taken by student prior to course: Coded zero for none, Coded 1 for one course, coded 2 for two courses, etc.
<b>OTHER</b>	Number of high school or college programming courses other than BASIC or COBOL previously taken by student prior to course: Coded zero for none, Coded 1 for one course, coded 2 for two courses, etc.
<b>PROGEXP</b>	Total number of high school or college programming courses previously taken by student prior to the course.

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>APPLE</b>	Pre-course experience in using an Apple computer: No experience coded zero, and any experience coded 1.
<b>MACINTOSH</b>	Pre-course experience in using a MACINTOSH computer: No experience coded zero, and any experience coded 1.
<b>IBM</b>	Pre-course experience in using an IBM-compatible computer: No experience coded zero, and any experience coded 1.
<b>OTHER</b>	Pre-course experience in using any other computer: No experience coded zero, and any experience coded 1.
<b>MICROEXP</b>	Pre-course length of time respondent has used a microcomputer: coded in total months, 0 for no previous usage, 1 for one month previous usage, 2 for two months previous usage, etc.



Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>SPREADSHEET</b>	Pre-course experience in using microcomputer spreadsheet package: No experience in using a microcomputer spreadsheet software coded 0, and any microcomputer spreadsheet software experience coded 1.
<b>WORDPROCESSING</b>	Pre-course experience in using microcomputer wordprocessing packages: No experience in using a microcomputer wordprocessing software coded 0, and any microcomputer wordprocessing software experience coded 1.
<b>HOME</b>	Best description of home computer usage prior to course: No computer used at home coded 0, computer at home used mostly for pleasure, e.g., video games coded 1, Computer at home used mostly for word processing and spread sheets coded 2, computer at home used mostly for writing programs coded 3, and computer at home mostly used for other reasons coded 4.
<b>WORK</b>	Best description of work computer usage prior to course: No computer used at work coded 0, computer at work used mostly for mostly for word processing and spread sheets coded 1, Computer at work used mostly for writing programs coded 2, and computer at home mostly used for other reasons coded 3.

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>REQUIRED</b>	This course is not a required course for student coded 0. This course is a required course for major coded 1.
<b>COMPINT</b>	Student is not interested in learning more about computers coded 0. Student is interested in learning more about computers coded 1.
<b>PROGINT</b>	Student is not interested in learning about computer programming coded 0. Student is interested in learning more about computer programming coded 1.
<b>HOURS WORK</b>	Number of hours of employment/work per week for each student. Coded to the nearest hour.
<b>ATTENDANCE</b>	Number of classes missed by students, coded percent of classes missed.

<b>Table 12 (cont.)</b>	
<b>List of Variables and their Coding Schemes Used in the Study</b>	
<b>Variable Label</b>	<b>Description and Coding</b>
<b>TEST1</b>	Program language prerequisite test 1 questions score, coded percent correct.
<b>TEST2</b>	Program language prerequisite test 2 questions score, coded percent correct.
<b>TEST3</b>	Program language prerequisite test 3 questions score, coded percent correct.
<b>PREREQ</b>	Total of program language prerequisite questions score, coded percent correct.

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>DEBUGS1</b>	Debugging score for test program one, question 1, coded 0 points for failure to locate error, 3 points located error but failed to correct, 8 points located and corrected error.
<b>DEBUGS2</b>	Debugging score for test program two, coded 0 points for failure to locate error, 9 points located error but failed to correct, 24 points located and corrected error.
<b>DEBUGS3</b>	Debugging score for test program three, coded 0 points for failure to locate error, 9 points located error but failed to correct, 24 points located and corrected error.
<b>DEBUGS4</b>	Debugging score for test program four, coded 0 points for failure to locate error, 15 points located error but failed to correct, 40 points located and corrected error.
<b>DEBUGS5</b>	Debugging score for test program five, coded 0 points for failure to locate error, 15 points located error but failed to correct, 40 points located and corrected error.
<b>LOCCOR</b>	Total score for all debugging test programs, coded total points received for locating and debugging program logic errors.

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
DEBUGT1	Time spent locating and correctly debugging question one's logic error, coded in minutes, unanswered questions coded maximum time (40 minutes).
DEBUGT2	Time spent locating and debugging question two's logic error coded in minutes, unanswered questions coded maximum time (40 minutes).
DEBUGT3	Time spent locating and debugging question three's logic error, coded in minutes, unanswered questions coded maximum time (40 minutes).
DEBUGT4	Time spent locating and debugging question four's logic error, coded in minutes, unanswered questions coded maximum time (40 minutes).
DEBUGT5	Time spent locating and debugging question five's logic error, coded in minutes, unanswered questions coded maximum time (40 minutes).
TIMECOR	Total time spent locating and debugging all test questions' logic errors coded in minutes, no questions answered coded maximum time (200 minutes).

Table 12 (cont.)	
List of Variables and their Coding Schemes Used in the Study	
Variable Label	Description and Coding
<b>GEFT SCORE</b>	Score on <u>Group Embedded Figures Test</u> , coded zero to 18.
<b>FIELD TYPE</b>	Degree of Field Dependence based upon the GEFT SCORE, coded: 0 Field Dependent for GEFT SCORE zero thru 8 inclusive, 1 Indeterminate for GEFT SCORE 9 thru 11 inclusive, and 2 Field Independent for GEFT SCORE 12 thru 18 inclusive.

## **RESEARCH FINDINGS**

### **Introduction**

In this chapter, the results of the research questions as outlined in Chapter II are discussed, beginning with a discussion of the data transcription tools and statistical software used to collect and analyze the data. This discussion is followed by a demographic profile of the students, descriptive statistics, the analysis of the prerequisite tests and debugging posttests, and presentation of the criteria used to select the covariates. The results of Analysis of Covariance for each research question are presented, and this section is concluded with a secondary analysis of each posttest program debugging task.

### **Data transcription and statistical software**

Q&A version 3.0 (Q&A User's Manual, 1988), a microcomputer data entry and database software package, was used to record the research data prior to transferring (exporting) it to SAS (SAS/STAT User's Guide, 1988), a statistical, software package. All descriptive statistics and statistical analysis of the research data were performed using SAS microcomputer version 6.03.

### Demographic Profile

Forty Robert Morris students enrolled in two sections of CI201 Business Programming (SECT1 & SECT2) and forty-five University of Pittsburgh students enrolled in two sections of CS4/007 BASIC Programming (SECT3 & SECT4) participated in this study. As shown in Table 14, the average age of all the participants was 21.2 years and represented all college undergraduate class levels, freshmen through seniors (see Tables 15 and 16). Males comprised 57.6% of the participants (see Table 13). With the exception of one group (SECT4) the proportion of males and females were approximately the same.

	SECT1	SECT2	SECT3	SECT4	ALL
<b>Male</b>	10 50%	12 60%	11 50%	16 69.5%	49 57.6%
<b>Female</b>	10 50%	8 40%	11 50%	7 30.5%	36 42.4%

	SECT1	SECT2	SECT3	SECT4	ALL
<b>Mean</b>	21.85	22.25	20.32	20.83	21.27
<b>Standard Deviation</b>	1.42	2.51	2.06	2.37	2.24
<b>Minimum</b>	20.00	20.00	18.00	19.00	18.00
<b>Maximum</b>	26.00	30.00	28.00	28.00	30.00



Table 15 Number of College Credits Taken Prior to Course					
	SECT1	SECT2	SECT3	SECT4	ALL
Mean	67.9	70.0	48.4	42.2	56.45
Standard Deviation	22.7	15.8	35.5	28.3	29.13
Minimum	15	47	13	8	8
Maximum	104	106	106	104	106

Table 16 Number of Students Classified by Collegiate Year					
	SECT1	SECT2	SECT3	SECT4	ALL
<b>FRESHMAN</b>	1 5%	0 0%	9 40.9%	8 34.7%	18 21.1%
<b>SOPHOMORE</b>	5 25%	5 25%	6 27.2%	9 39.1%	25 29.4%
<b>JUNIOR</b>	12 60%	14 70%	4 18.1%	3 13.0%	33 38.8%
<b>SENIOR</b>	2 10%	1 5%	4 18.1%	2 8.6%	9 10.6%

As shown in Table 17, only one (1) student who participated in the experiment was declared a Computer Information Science or Computer Science major. This low proportion of computer-related majors was expected for several reasons. First, many Computer Information Science majors at Robert Morris College have historically transferred the equivalent of CI201 Business Programming credits from local community colleges, or otherwise would

have taken the course in the FALL semester rather than the Winter semester when the experiment was conducted. Second, the majority of the students enrolled in CI201 Business Programming at Robert Morris College were Accounting and Finance majors, and this programming course was required for their respective major. Third, the University of Pittsburgh's Computer Science majors were discouraged from enrolling in the CS4/007 BASIC course. This course was specifically designed to be a service course for non-Computer Science Majors.

	MAJOR				
	SECT1	SECT2	SECT3	SECT4	ALL
<b>Undeclared</b>	0 0.0%	0 0.0%	7 31.8	7 30.4%	14 16.4%
<b>Social Science</b>	0 0.0%	0 0.0%	0 0.0	1 4.3%	1 1.1%
<b>Business</b>	19 95%	20 100%	2 .9%	4 17.3%	45 52.9%
<b>Mathematics</b>	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%
<b>Engineering</b>	0 0.0%	0 0.0%	4 18.1%	2 8.7%	6 7.0%
<b>Education</b>	0 0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%
<b>Computer Science</b>	1 5%	0 0.0%	1 4.5%	3 13%	4 4.7%
<b>Natural Sciences</b>	0 0.0%	0 0.0%	8 36.3%	6 26%	14 16.4%

As shown in Table 17, the majority of the participants of the study were business majors (52.9%). Natural science majors (16.4%) or undeclared majors (16.4%) represented the majority of the remaining participants. As reported in Table 18, approximately seventy-nine percent (78.8%) reported that this programming course was required for their major. Seventy-four (74.2%) of the participants reported some interest in learning computers prior to the course and sixty-two (62.4%) reported some interest in learning how to program a computer prior to enrolling in the course.

<b>Table 18</b>					
<b>Prior Computer and Programming Interest</b>					
<b>Required Course</b>	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
<b>No</b>	2 10%	4 20%	6 27.2%	6 26%	18 21.2%
<b>Yes</b>	18 90%	16 80%	16 72.8%	17 74%	67 78.8%
<b>Interest in Computers</b>					
<b>None</b>	3 15%	5 25%	8 36.3%	6 26%	22 25.8%
<b>Some</b>	17 85%	15 75%	14 64.7%	17 74%	63 74.2%
<b>Interest in Programming</b>					
<b>None</b>	6 30%	9 45%	8 36.3%	9 39.1%	32 37.6%
<b>Some</b>	14 70%	11 55%	12 73.7%	14 61.9%	53 62.4%

Only four (4) students had no microcomputer experience prior to the study and the average microcomputer usage experience was slightly over two years (see Table 19). Twenty-seven percent (27%) of the participants had successfully completed a previous course in the BASIC programming language, five percent (5%) had taken a previous course in PASCAL, and no participants had taken a previous COBOL course (see Tables 20 and 21).

Table 19 Microcomputer Usage Experience (in months)					
	SECT1	SECT2	SECT3	SECT4	ALL
Mean	32.65	38.65	19.09	22.96	27.93
Standard Deviation	16.47	16.47	25.23	29.03	25.47
Minimum	6.00	7.00	0.00	0.00	0.00
Maximum	66.00	96.00	84.00	96.00	96.00

<b>APPLE</b>	SECT1	SECT2	SECT3	SECT4	ALL
No Experience	15 75%	14 70%	16 72.7%	17 73.9%	62 72.9%
Experience	5 25%	6 30%	6 33.3%	6 32.1%	23 33.1%
<b>MACINTOSH</b>					
No Experience	19 95%	20 100%	17 77.2%	15 65.2%	71 83.5%
Experience	1 5%	0 0.0%	5 32.8%	8 34.8%	14 16.5%
<b>IBM Compatible</b>					
No Experience	0 0.0%	0 0.0%	10 45.4%	8 34.7%	18 21.1%
Experience	20 100%	20 100%	12 54.6%	15 63.3%	67 78.2%
<b>Other Computers</b>					
No Experience	19 95%	18 90%	21 95.4%	20 90.9%	78 91.7%
Experience	1 5%	2 10%	1 4.6%	3 9.1%	7 8.3%

	SECT1	SECT2	SECT3	SECT4	ALL
Mean	.45	.60	.36	.35	.44
Standard Deviation	.60	.75	.58	.49	.61
Minimum	0	0	0	0	0
Maximum	2	2	2	1	2

<b>Table 22 Previous Courses in Programming Languages Classified by Language</b>					
<b>BASIC</b>	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
No courses	15 75%	14 70%	16 72.7%	17 73.9%	62 72.9%
At least one course or prior experience	5 25%	6 30%	6 27.3%	6 26.1%	23 27.1%
<b>COBOL</b>					
No Courses	20 100%	20 100%	22 100%	23 100%	85 100%
At least one course or prior experience	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%
<b>OTHER COURSES</b>					
No Courses	16 80%	14 70%	20 90.9%	21 91.3%	81 95.2%
At least one course or prior experience	4 20%	6 30%	2 9.1%	2 8.7%	4 4.8%

As shown in Table 23, approximately seventy-one percent (70.6%) of the participants had previous experience in the use of wordprocessing software packages, e.g., WordPerfect, and approximately fifty-two percent (51.8%) had previous experience in the use of spreadsheet software packages, e.g., LOTUS. As shown in Table 24, approximately fifty-one percent (51.8%) of the participants owned a home personal

computer, which they predominately used for wordprocessing or to prepare spreadsheets. Only twenty-three percent of the participants had previously used a computer at work.

Computer work experience was generally limited to wordprocessing and the preparation of spreadsheets.

<b>Table 23</b>					
<b>Other Computer Software Experiences Prior to Study</b>					
<b>Spread Sheet</b>	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
No Experience	1 5%	2 10%	19 86.3%	19 82.6%	41 48.2%
Experience	19 95%	18 90%	3 16.7%	4 17.4%	44 51.8%
<b>Word Processing</b>					
No Experience	4 20%	3 15%	6 27.2%	12 52.1%	25 29.4%
Experience	16 80%	17 85%	16 72.8%	11 51.9%	60 70.6%
<b>Other Formal Training</b>					
No Training	18 90%	19 95%	21 95.4%	22 95.6%	80 94.1%
Some Training	2 10%	1 5%	1 4.6%	1 4.4%	5 5.9%

Table 24					
Prior Home and Work Computer Usage					
HOMCOMP	SECT1	SECT2	SECT3	SECT4	ALL
No Usage	12 60%	7 35%	11 50%	11 47.8%	41 48.2%
Pleasure	0 0.0%	4 20%	2 9%	5 21.7%	11 12.9%
Word Processing/ Spreadsheets	8 40%	7 35%	10 45.4%	5 21.7%	30 35.2%
Programming	0	2 10%	1 4.5%	2 8.6%	5 5.8%
Other	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%
WORKCOMP					
No Usage	16 80%	15 75%	18 81.8%	19 82.6%	65 76.4%
Word Processing/ Spreadsheets	4 20%	3 15%	3 13.6%	4 17.4%	14 16.4%
Programming	0 0.0%	0 0.0%	1 4.5%	0 0.0%	1 1.1%
Other	0 0.0%	2 10%	0 0.0%	0 0.0%	2 2.3%

Field dependence, as defined previously, "refers to a consistent mode of approaching the environment in analytical as opposed to global terms. It denotes the ability to articulate figures as discrete from their backgrounds and an ability from disembedding contexts" (Messick, 1977, p. 14). The Group Embedded Figures Test (GEFT) was used to measure the level of field dependence.



The mean score on the GEFT was 11.58 with a standard deviation of 4.35 (see Table 25). These results were similar to norming studies used to develop the test (Witkin, et al. 1971, p. 28, mean = 11.4, standard deviation = 4.15). Using the mean score of the GEFT and the standard deviation of approximately +/- .5, the participants were assigned to one of three groups: Field Dependent (GEFT score from zero to ten inclusive), Indeterminate (GEFT score eleven through thirteen inclusive), or Field Independent (GEFT score from fourteen to eighteen inclusive). Field Dependent individuals represented 33% of the participants. Field Independent represented 41% of the participants and 26% of the participants were classified as Indeterminate (see Table 26).

	SECT1	SECT2	SECT3	SECT4	ALL
Mean	12.05	11.95	10.68	11.70	11.58
Standard Deviation	3.93	3.69	4.68	5.01	4.35
Minimum	5.00	6.00	3.00	2.00	2.00
Maximum	17.00	18.00	10.00	18.00	18.00

	SECT1	SECT2	SECT3	SECT4	ALL
<b>Field Independent</b>	6 30%	7 35%	9 40.9%	6 26%	28 33.0%
<b>Field Dependent</b>	6 30%	4 20%	6 27.3%	6 26%	22 25.9%
<b>Indeterminant</b>	8 40%	9 45%	7 31.8%	11 48%	35 41.1%

The threats to internal and external validity caused by the selection bias of intact groups has been previously discussed. Comparisons of the differences in background and demographic data found between debugging treatment and control groups within each programming language, COBOL and BASIC, were statistically insignificant ( $p > .05$ ). However, slight demographic and background differences were found between the two schools. These differences between the participants of the two different schools included: (a) the COBOL participants were approximately one (1) year older (see Table 14), (b) the COBOL participants were predominately juniors and seniors; whereas, the BASIC participants were predominately freshman and sophomores (see Table 16), (c) all of the COBOL participants were business majors; whereas, the BASIC participants represented a variety of majors (see Table 17), (d) the COBOL participants had more prior experience in using a microcomputer and in the use of wordprocessing and spreadsheet packages (see

Tables 19 and 20), and (e) more COBOL students worked at outside employment during the study than BASIC students (see Tables 27 and 28). None of these differences were found to significantly correlate with any dependent variable ( $p > .05$ ).

<b>Table 27</b>					
<b>Hours Worked per Week by the Participant During Study</b>					
	SECT1	SECT2	SECT3	SECT4	ALL
<b>Mean</b>	12.75	16.1	4.05	4.04	8.93
<b>Standard Deviation</b>	10.04	12.34	8.41	8.36	11.03
<b>Minimum</b>	0	0	0	0	0
<b>Maximum</b>	30	40	30	28	30

<b>Table 28</b>					
<b>Number of Participants Who Worked During the Study</b>					
	SECT1	SECT2	SECT3	SECT4	ALL
<b>Students who did not work</b>	4 20%	6 30%	16 72.7%	17 73.9%	43 50.5%
<b>Employed Students</b>	16 80%	14 70%	6 27.3%	6 22.1%	41 48.5%

### Results of the program language prerequisite tests

The ability to locate and correct a logic error in a computer program required the mastery of syntax rules and the development of algorithmic skills. Data gathered from three program language prerequisite tests for each programming language were used to measure the student's mastery of prerequisite programming skills. Standardized test items designed for the specific versions of MicroFocus's COBOL and MicroSoft's QuickBASIC were not available. Three objective tests were developed for each programming language.

The results of the prerequisite tests used in the study were analyzed by the Test Analysis program provided by the University of Pittsburgh's Office of Measurement and Evaluation. The results of this analysis are shown in Table 29. A total of 123 test items were administered to the COBOL participants and a total of 104 test items were administered to BASIC participants on three different test dates. The mean number of correct answers on the COBOL prerequisite tests was 81.6 with a KR-20 reliability coefficient of .93. The mean number of correct answers on the BASIC prerequisite tests was 63 with a KR-20 reliability coefficient of .86.

The difference in the number of test items between each programming language prerequisite test was attributable to syntactical differences between the programming languages.

All conceptual and algorithmic topics on the prerequisite test between languages were the same.

	COBOL				BASIC			
	TEST 1	TEST 2	TEST 3	ALL	TEST 1	TEST 2	TEST 3	ALL
<b>No. of Test Items</b>	50	60	23	123	41	46	17	104
<b>Mean Test Items Correct</b>	32.0	38.1	12.0	81.6	27.9	25.9	9.40	63.0
<b>Standard Deviation</b>	6.36	8.77	4.82	18.0	4.73	5.67	2.33	11.7
<b>KR-20</b>	.82	.87	.84	.93	.71	.75	.48	.86
<b>No. of Test Items with Negative Point Biserial</b>	1	1	0	2	1	3	2	6
<b>No. of Test Items with Difficulty Index &gt; .90</b>	5	4	0	9	7	1	1	9
<b>No. of Test Items with Difficulty Index &lt; .10</b>	1	0	0	1	0	1	0	1

As shown in Table 30, the mean percentage test score for all four treatment groups was approximately 61% with a standard deviation of 12%. No significant differences in the mean percentage test score between groups were found. The

mean prerequisite test percentage for all four sections declined from the highest mean prerequisite test percentage on TEST1 to the lowest mean prerequisite test percentage on TEST3 (see Table 31). These results were expected since most programming students have greater difficulty mastering algorithmic concepts presented on the latter tests, rather than syntactical concepts presented on the earlier tests. Descriptive statistics for each individual prerequisite test are shown in Tables 32, 33 and 34.

	SECT1	SECT2	SECT3	SECT4	ALL
Mean	59.30	63.70	61.77	59.65	61.07
Standard Deviation	12.59	14.21	10.49	11.02	12.00
Minimum	38.00	41.00	40.00	45.00	38.00
Maximum	82.00	92.00	81.00	80.00	92.00

	TEST 1	TEST 2	TEST 3	LOCCOR
SECT1	64.40%	59.70%	47.35%	59.30%
SECT2	62.00%	65.80%	52.40%	63.70%
SECT3	68.41%	57.77%	55.64%	61.77%
SECT4	67.43%	54.61%	54.22%	59.65%

<b>Table 32</b>					
<b>Prerequisite Test 1 Percentage Score by Treatment Group</b>					
	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
<b>Mean</b>	64.40	62.00	68.41	67.43	65.69
<b>Standard Deviation</b>	11.83	16.07	11.97	11.41	12.90
<b>Minimum</b>	42.00	26.00	37.00	51.00	26.00
<b>Maximum</b>	82.00	84.00	90.00	90.00	90.00

<b>Table 33</b>					
<b>Prerequisite Test 2 Percentage Score by Treatment Group</b>					
	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
<b>Mean</b>	59.70	65.80	57.77	54.61	59.26
<b>Standard Deviation</b>	13.41	16.87	11.97	13.12	14.24
<b>Minimum</b>	30.00	33.00	35.00	30.00	30.00
<b>Maximum</b>	83.00	95.00	76.00	76.00	95.00

<b>Table 34</b>					
<b>Prerequisite Test 3 Percentage Score by Treatment Group</b>					
	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
<b>Mean</b>	47.35	52.40	55.64	54.22	52.54
<b>Standard Deviation</b>	19.28	23.35	13.55	14.65	17.87
<b>Minimum</b>	17.00	17.00	29.00	29.00	17.00
<b>Maximum</b>	83.00	100.0	76.00	82.00	100.0

Descriptive statistics of the program debugging test

Two program debugging posttests were developed for each computer language (Appendices J and K). The same test was administered to both the interactive and the traditional debugging sections in each respective programming language. This test was designed to measure: (a) the ability to locate and correct a program logic error (LOCCOR) and (b) the time to locate and correct a program logic error (TIMECOR).

An adequate test was unavailable to test the students ability to debug logic errors in a COBOL and BASIC program. The program debugging posttest used in this study was first administered to a pilot group of students and subsequently minor revision were made. The test was then administered to the participants of this study.

Table 35 Total Computer Debugging Test Score LOCCOR (Maximum points = 136)					
	SECT1	SECT2	SECT3	SECT4	ALL
Mean	78.8	94.8	75.3	93.1	85.53
Standard Deviation	51.3	41.4	43.6	43.8	45.14
Minimum	0	24	0	0	0
Maximum	136	136	136	136	136



<b>Table 36</b> <b>Total Debugging Time to Locate and Correct an Error</b> <b>TIMECOR</b> <b>(Maximum Time = 200 minutes)</b>					
	<b>SECT1</b>	<b>SECT2</b>	<b>SECT3</b>	<b>SECT4</b>	<b>ALL</b>
<b>Mean</b>	106.3	104.3	117.5	101.8	107.5
<b>Standard Deviation</b>	61.0	45.2	42.6	42.6	47.6
<b>Minimum</b>	25	38	23	49	25
<b>Maximum</b>	200	167	200	200	200

As shown in Table 35, the interactive debugging treatment groups, SECT2 AND SECT4, had a total mean program debugging test score (LOCCOR) of 94.8 and 93.1 respectively. The traditional debugging groups, SECT1 and SECT3, had a total mean program debugging test score (LOCCOR) of 78.8 and 75.3, respectively. Overall, the effect size between the interactive and the traditional debugging total mean test scores (LOCCOR) was .38 standard deviations. The effect size was slightly larger for BASIC programming language groups, SECT3 and SECT4, than the COBOL programming language groups, SECT1 and SECT2.

As shown in Table 36, the differences found in the total mean time to locate and correct a program logic error (TIMECOR) was not as large as the differences found between the total mean debugging test scores (LOCCOR). The distribution of the total mean debugging times was bimodal

and positively-skewed. This distribution was attributable to the inherent nature of time-oriented data and the coding of incorrect posttest program sections with a maximum time value.

The interactive program debugging groups, SECT2 and SECT4, had a total mean debugging time (TIMECOR) of 104.3 and 101.8 minutes respectively (see Table 36). The traditional program debugging groups, SECT1 and SECT3, had a total mean debugging time (TIMECOR) of 106.3 and 117.5, respectively. Overall, the effect size between the interactive and the traditional total mean debugging time (TIMECOR) was approximately .23 standard deviations. The effect size was larger for the BASIC programming treatment groups, SECT3 and SECT4, than the COBOL programming treatment groups, SECT1 and SECT2 (see Table 36).

Each program debugging posttest consisted of five programs. The first program contained one simple logic error, e.g., failure to execute a statement within a loop. The second and third program contained a more difficult logic error, e.g., incorrect sequence of statements. The fourth and the fifth programs contained two logic errors that interacted. Descriptive statistics for LOCCOR and TIMECOR analyzed for each individual debugging test program are presented in Tables 37 and 38.

<b>Table 37</b>						
<b>Debugging Mean Test Score by Question and by Section</b>						
	<b>1</b> max=8	<b>2</b> max=24	<b>3</b> max=24	<b>4</b> max=40	<b>5</b> max=40	<b>Total</b> max=136
<b>SECT1</b>	6.40	18.00	14.40	20.00	20.00	74.8
<b>SECT2</b>	4.80	18.00	17.95	30.00	24.00	94.8
<b>SECT3</b>	6.91	15.00	20.73	23.59	9.09	75.3
<b>SECT4</b>	6.96	18.78	21.91	29.78	15.65	93.1
<b>ALL</b>	6.31	17.44	18.91	25.93	16.94	85.53

<b>Table 38</b>						
<b>Debugging Mean Test Time in Minutes by Question</b>						
	<b>1</b> max=40	<b>2</b> max=40	<b>3</b> max=40	<b>4</b> max=40	<b>5</b> max=40	<b>Total</b> max=200
<b>SECT1</b>	15.75	16.60	21.50	24.60	27.90	106.35
<b>SECT2</b>	22.85	18.15	16.40	20.10	25.35	104.30
<b>SECT3</b>	20.14	26.18	11.82	21.73	35.41	117.55
<b>SECT4</b>	19.30	19.87	11.78	19.65	31.39	101.87
<b>ALL</b>	19.52	20.33	15.16	21.46	30.19	107.55

An item analysis of the debugging test is presented in the Table 39. As expected, questions four and five demonstrated the greatest ability to differentiate student's program debugging skills.

	1 max=8	2 max=24	3 max=24	4 max=40	5 max=40
<b>Mean</b>	6.31	17.44	18.91	25.93	16.94
<b>Difficulty Index</b>	.788	.729	.788	.682	.424
<b>Discrimination Index</b>	.443	.612	.612	.927	.965
<b>Point Biserial Correlation</b>	.464	.623	.634	.845	.751

The objective of this study was to investigate the instructional effectiveness of two different instructional methods used to teach students to correct logic errors contained in a computer program. There are three interrelated issues to be considered in the evaluation of these program debugging tools: (a) the instructional presentation qualities of the debugging tool, (b) the adoption of the tool, and (c) the skills in using the debugging tool.

The "instructional presentation qualities" of the debugging tool includes those issues that enable a student to better: (a) understand and recognize the types of program logic errors and (b) detect the symptoms, i.e., program output, and causes of program logic errors. "Presentation qualities," as defined in this study, represent lower level cognitive objectives, such as knowledge, comprehension and interpretation.

While a computer debugging tool may offer benefits beyond understanding program logic errors better, students may not actually use the debugging tool while programming. The "adoption qualities" of computer debugging tools, as defined in this study, include those issues relating to the student's choice to use the tool. "Adoption qualities" are more affective in nature than cognitive.

"Skills in using the tool," as defined in this study, include various higher level cognitive objectives such as application and analysis. The ability to use a particular program debugging tool to solve a new and different debugging problem may be the best measure of the practical significance of the computer debugging tool.

While the dependent variables, LOCCOR and TIMECOR, may provide some evidence to assess the effectiveness of the debugging tool as a presentation tool, these dependent variables do not necessarily provide sufficient evidence to assess the "adoption and skill qualities" of the computer debugging tool. Therefore, it is important to determine whether the student actually did use the computer debugging tool. In addition, the effectiveness of the program debugging tool when it was applied to a new computer debugging task needs to be addressed.

A self-reporting instrument was included with each debugging test program. The student was asked to report on each debugging tool used to debug a particular test program

and to rate the value of the debugging tool used. The "Value of Tool" rating scale ranged from: one (1) not used, two (2) slight value, to five (5) very valuable. Values were always reported, even if the student did not successfully locate and correct the program logic error.

The following tables report various descriptive statistics gathered from the debugging tool worksheet prepared by students at the end of each program section of the program debugging test. "Percent used" represented the percentage of students who reported to have used a particular tool during the posttest for a particular program. "Mean Value" represented the statistical mean of the "Value of Tool" rating scale. The correlation coefficient measured the association between the reported value of the tool used and the student's test score for that particular program section. Correlation coefficients that were not statistically significant ( $p > .05$ ) were reported as zero in the following tables. Tables 40 thru 43 presents descriptive statistics for each instructional section concerning the reported usage and the value of the computer debugging tools used during the posttest.

<b>Table 40</b> <b>Descriptive Statistics of Debugging Tool Used</b> <b>by Question for SECT3</b> <b>(BASIC Traditional Debugging)</b>					
	1	2	3	4	5
<b>Reviewed Prog. Code</b>					
Percent Used	100%	100%	100%	100%	100%
Mean Value	4.09	3.86	4.00	3.45	2.59
Standard Deviation	1.01	1.12	1.38	1.73	1.91
Correlation Coeff.	.30	.83	.49	.47	.46
<b>Reviewed Outputs</b>					
Percent Used	87.4%	73.7%	73.7%	68.2%	46.5%
Mean Value	3.40	3.45	3.50	3.36	2.68
Standard Deviation	1.01	1.56	1.76	1.83	1.91
Correlation Coeff.	.00	.00	.00	.45	.00
<b>Displays/Prints</b>					
Percent Used	4.5%	31.8%	9.1%	13.5%	13.5%
Mean Value	1.09	1.72	1.36	1.22	1.40
Standard Deviation	.426	1.27	1.17	.869	1.09
Correlation Coeff.	.00	.00	.00	.00	.00

<b>Table 41</b> <b>Descriptive Statistics of Debugging Tool Used</b> <b>by Question for SECT1</b> <b>(COBOL Traditional Debugging)</b>					
	1	2	3	4	5
<b>Reviewed Prog. Code</b>					
Percent Used	100%	100%	95%	95%	100%
Mean Value	4.20	4.15	4.10	4.00	3.00
Standard Deviation	0.95	1.03	1.25	1.25	1.86
Correlation Coeff.	.51	.65	.40	.57	.60
<b>Reviewed Outputs</b>					
Percent Used	75%	80%	80%	75%	55%
Mean Value	3.35	3.55	3.75	3.50	2.85
Standard Deviation	1.59	1.57	1.65	1.76	1.89
Correlation Coeff.	.00	.58	.69	.64	.00
<b>Display/Print</b>					
Percent Used	5%	0%	5%	0%	0%
Mean Value	1.10	1.00	1.20	1.00	1.00
Standard Deviation	0.44	0.00	.894	0.00	0.00
Correlation Coeff.	.00	.00	.00	.00	.00



**Table 42**  
**Descriptive Statistics of Debugging Tool Used**  
**by Question for SECT4**  
**(BASIC Interactive Debugging)**

	1	2	3	4	5
<b>Reviewed Prog. Code</b>					
Percent Used	100%	91.6%	100%	95.7%	78.3%
Mean Value	3.43	3.39	3.69	3.69	3.00
Standard Deviation	1.30	1.49	1.36	1.42	1.62
Correlation Coeff.	.00	.42	.00	.00	.44
<b>Reviewed Outputs</b>					
Percent Used	82.6%	82.6%	82.6%	78.3%	65.3%
Mean Value	3.13	3.47	3.47	3.39	2.86
Standard Deviation	1.42	1.53	1.59	1.55	1.65
Correlation Coeff.	.49	.00	.41	.53	.50
<b>Displays/Prints</b>					
Percent Used	17.4%	13.0%	13.0%	8.6%	17.4%
Mean Value	1.34	1.39	1.34	1.13	1.30
Standard Deviation	.83	1.15	1.02	0.45	0.92
Correlation Coeff.	.00	.00	.00	.00	.00
<b>Stepping</b>					
Percent Used	60.9%	65.3%	62.2%	69.6%	75.3%
Mean Value	2.73	2.73	2.78	2.95	2.65
Standard Deviation	1.68	1.54	1.67	1.58	1.92
Correlation Coeff.	.00	.00	.00	.00	.00
<b>Break Points</b>					
Percent Used	44.8%	39.2%	26.1%	21.8%	26.1%
Mean Value	1.60	1.91	1.86	1.52	1.78
Standard Deviation	.98	1.41	1.60	1.16	1.44
Correlation Coeff.	.00	.00	.00	.00	.00
<b>Query Variables</b>					
Percent Used	34.8%	56.6%	44.5%	61.2%	39.2%
Mean Value	1.86	2.52	2.30	2.17	2.13
Standard Deviation	1.28	1.59	1.63	1.64	1.57
Correlation Coeff.	.00	.00	.00	.00	.00

**Table 43**  
**Descriptive Statistics of Debugging Tool Used**  
**by Question for SECT2**  
**(COBOL Interactive Debugging)**

	1	2	3	4	5
<b>Reviewed Prog. Code</b>					
Percent Used	100%	100%	100%	90%	100%
Mean Value	4.10	4.20	4.20	4.00	2.90
Standard Deviation	1.02	1.05	1.15	1.48	1.86
Correlation Coeff.	.39	.42	.61	.71	.85
<b>Reviewed Outputs</b>					
Percent Used	80%	75%	70%	75%	55%
Mean Value	3.85	3.60	3.75	3.40	1.20
Standard Deviation	1.53	1.63	1.61	1.78	1.83
Correlation Coeff.	.00	.00	.63	.00	.77
<b>Displays/Prints</b>					
Percent Used	5%	5%	5%	5%	5%
Mean Value	1.10	1.05	1.30	1.60	1.20
Standard Deviation	0.30	.22	.92	1.46	0.89
Correlation Coeff.	.00	.00	.00	.00	.00
<b>Stepping</b>					
Percent Used	35%	35%	50%	50%	50%
Mean Value	2.20	2.15	2.10	2.60	2.75
Standard Deviation	1.54	1.66	1.29	1.72	2.02
Correlation Coeff.	.00	.00	.00	.00	.72
<b>Break Points</b>					
Percent Used	20%	20%	20%	25%	20%
Mean Value	1.40	1.55	1.20	1.65	1.65
Standard Deviation	0.94	1.19	.52	1.30	1.46
Correlation Coeff.	.00	.00	.00	.00	.00
<b>Query Variables</b>					
Percent Used	20%	25%	10%	30%	30%
Mean Value	1.75	1.90	1.45	2.05	2.00
Standard Deviation	1.44	1.55	1.23	1.70	1.65
Correlation Coeff.	.00	.00	.00	.00	.50

### Interpretations

As shown in Tables 40 thru 43, "Reviewing the Program's Source Code" was reported as the most frequently used and the most valuable computer debugging tool for all posttest programs and experimental sections. "Reviewing the incorrect printed outputs" was reported as the second most frequently used and valuable computer debugging tool for all posttest programs and experimental sections. The students who participated in the traditional debugging sections rarely used DISPLAY and the PRINT verbs to debug their test programs.

### Selection of Covariates

Two variables, programming experience (PROGEXP) and prerequisite knowledge (PREREQ), were expected to covary with each dependent variable (LOCCOR and TIMECOR) and, thus, must be taken into account to minimize confounding the results of the study. Programming experience represented programming knowledge and abilities previously acquired in other programming courses or by professional experience. In addition, programming experience may have included other programming languages other than those being studied. PROGEXP was coded as the number of high school or college computer programming language courses taken prior to the study.

Prerequisite knowledge, on the other hand, related to

the specific rules of syntax, program construction and algorithmic development for the particular programming language being studied. Prerequisite knowledge was measured by the total percentage score of the program language prerequisite tests.

The analysis of the covariates proceeded in two steps. First, a Pearson correlation was performed on PROGEXP and PREREQ to measure their relationship with each of two dependent variables: LOCCOR and TIMECOR. The results of this analysis are shown in Tables 44 and 45.

Table 44 Pearson Correlation Analysis of the Variable PROGEXP (the number of previous programming courses)	
Variable	Pearson Correlation Coefficient
LOCCOR	0.08
TIMECOR	-0.17
PREREQ	0.21 *

\*  $p < .05$

Table 45 Pearson Correlation Analysis of the Variable PREREQ (total percentage prerequisite test scores)	
Variable	Pearson Correlation Coefficient
LOCCOR	0.43 *
TIMECOR	-0.48 *
PROGEXP	0.21 *

\*  $p < .05$

A Pearson Correlation Analysis also was performed on other demographic and descriptive data with the dependent variables, LOCCOR and TIMECOR. No other significant correlations were found between the demographic and descriptive data and the dependent variables ( $p > .05$ ).

Second, a pooled within-groups correlation was performed on PROGEXP and PREREQ to measure their relationship with each of two dependent variables: LOCCOR and TIMECOR. The Pearson correlation measures the degree of association between any two variables, e.g., PROGEXP and PREREQ, using the data from all of the participants of the study. On the other hand, the pooled within-groups correlation measures the degree of association between any two variables by summing and weighting (sum of the squares of each variable) each Pearson correlation coefficient for each individual group or cell, e.g., COBOL field dependent individuals who used the interactive program debugger.

A pooled within-groups correlation coefficient of .30 or greater will generally reduce the error term of a given factorial model to a degree that one can recommend the use of the Analysis of Covariance model (ANCOVA), rather than the Analysis of Variance model (ANOVA).

The pooled within-groups correlation coefficients found between the variable PROGEXP and the dependent variables, LOCCOR and TIMECOR, were .107 and  $-.201$ , respectively. The pooled within-groups correlation coefficients found between

the variable PREREQ and the dependent variables, LOCCOR and TIMECOR, were .416 and -.432, respectively.

Based upon the results of the Pearson correlation analysis and pooled within-groups correlation analysis, the variable PREREQ was retained as a covariate and PROGEXP was rejected. PREREQ was retained a covariate for two reasons: (a) it was significantly correlated ( $p < .05$ ) with LOCCOR or TIMECOR and (b) both pooled within-groups correlation coefficients measuring the association between the dependent variables, LOCCOR and TIMECOR, and PREREQ exceeded .30.

#### Analysis of Covariance

The design of this study was a posttest-only, completely between-subjects, fully-crossed factorial involving three independent variables: (a) debugging treatment, (b) programming language, and (c) field dependence. A quasi-experimental strategy was used to study the effects of these independent variables on two dependent variables: (a) ability to locate and correct a program logic error (LOCCOR) and (b) the amount of time to successfully locate and correct a program logic error (TIMECOR).

The first dependent variable: ability to locate and correct a program logic error (LOCCOR), was analyzed in a 2 X 2 X 3 SAS General Linear Procedure (GLM-type) ANCOVA. A SAS GLM analysis-of-covariance, homogeneity-of-slopes model

was conducted. The GLM procedure was used to take in account the fact that the treatment groups were unbalanced (SAS/STAT User's Guide, 1988). The group sizes of this study varied from 20 to 23 participants.

The GLM procedure used the least-squares means (LSMs) approach to handle unbalanced designs. "LSMs are simply estimators of the class or subclass marginal means that would be expected had the design been balanced" (SAS/STAT User's Guide, 1988, p. 564).

The GLM-type ANCOVA model also adjusted for differences found in the dependent variable (LOCCOR) due to the covariate variable prerequisite programming knowledge (PREREQ). The use of the covariate, PREREQ, reduced the error term of the model from 2098.94 (ANOVA) to 1764.84.

Because of the difficulties in assessing the actual assumption of equal-slopes, a SAS GLM analysis-of-covariance, separate-slopes model was also conducted. The results of this analysis were comparable to the reported results of the homogeneity-of-slopes model.

The results of the GLM ANCOVA will assist in answering Research Questions 1 and 2. The results of the ANOVA model and the ANCOVA model are presented in Tables 46 thru 48.

Table 46					
General Linear Model Procedure					
LOCCOR					
(ability to locate and correct logic errors)					
Source of Variation	DF	Sum of Square	Mean Square	F Value	Pr>F
Model	12	44116.4	3676.3	2.08	.0287*
Error	72	123068.7	1764.8		
Corrected Total	84	171185.1			

\* p&lt;.05

Table 47					
General Linear Model Procedure (ANOVA)					
LOCCOR					
(ability to locate and correct logic errors)					
Source of Variation	DF	TYPE I SS	Mean Square	F Value	Pr>F
LANGUAGE	1	121.9	121.9	0.07	0.793
TREATMENT	1	6093.6	6093.6	3.45	0.067
LANGUAGE*TREATMENT	1	16.5	16.5	0.01	0.923
GEFTCD	2	9163.5	4596.7	2.60	0.080
LANGUAGE*GEFTCD	2	1547.8	773.9	0.44	0.646
TREATMENT*GEFTCD	2	240.4	120.2	0.07	0.934
LANGUAGE*TREATMENT*GEFTCD	2	748.2	374.1	0.21	0.809
PREREQ	1	26154.1	26154.1	14.82	0.003

\* p&lt;.05



Table 48

**General Linear Model Procedure (ANCOVA)  
LOCCOR  
(ability to locate and correct logic errors)**

Source of Variation	DF	TYPE III SS	Mean Square	F Value	Pr>F
LANGUAGE	1	85.0	85.0	0.05	0.826
TREATMENT	1	4022.3	4022.3	2.28	0.135
LANGUAGE*TREATMENT	1	335.1	335.1	0.19	0.664
GEFTCD	2	3683.6	1841.8	1.04	0.357
LANGUAGE*GEFTCD	2	504.4	252.2	0.14	0.867
TREATMENT*GEFTCD	2	654.9	327.4	0.17	0.831
LANGUAGE*TREATMENT *GEFTCD	2	388.7	194.3	0.11	0.895
PREREQ	1	26154.1	26154.1	14.82	0.003

\*p&lt;.05

Type I SS is sometimes referred to the sequential sums of squares. Type I SS gives the between-group sum of the squares and will add up to the total of the model sum of the squares. Type III sum of the squares is sometimes referred to as partial sums of squares. Type III SS represents the Type I SS after adjusting for the covariate (PREREQ). Both Type I SS and Type III represents total sum of the squares across groups (SAS/STAT User's Guide, 1988).

**Interpretations:**

There were no statistically significant interactions among the three independent variables: LANGUAGE, TREATMENT and GEFTCD (field dependence) and the dependent variable

LOCCOR (the ability to locate and correct a program logic error). There were no statistically significant main effects found for any of the independent variable: LANGUAGE, TREATMENT and GEFTCD and the dependent variable LOCCOR.

The second dependent variable, time to locate and correct a program logic error (TIMECOR), was analyzed in separate 2 X 3 SAS General Linear Procedure (GLM-type) ANCOVAs with each model conducted for each programming language. The variable prerequisite programming knowledge (PREREQ) was used as a covariate. Due to the nature of program development process, interpretive versus compiled, there was no reason to expect that a relationship existed between the amount of time it takes to debug a BASIC program will be different for a COBOL program.

Using the procedures previously used to analyze the dependent variable LOCCOR, a SAS GLM analysis-of-covariance, separate-slopes model was used to analyze the dependent variable TIMECOR. A logarithmic transformation was performed on the raw time data before the GLM ANCOVA was conducted.

As previously mentioned, the distribution of the total mean debugging times was bimodal and positively-skewed due to the inherent nature of time-oriented data and the coding of incorrect program sections with a maximum time value. "There are three major reasons for using transformations: 1) To achieve homogeneity of error variance. 2) To achieve normality of treatment-level distributions (or within-cell

distributions). 3) To obtain additivity of treatment effects" (Kirk, R., 1968, p.63).

While multiple methods for the transformation of raw data exist, the logarithmic transformation method was preferred since: 1) the dependent variable (TIMECOR) was a measure of a reaction time and 2) and the raw data were positively-skewed (Kirk, R., 1968, p. 65).

The results of these two GLM ANCOVAs will assist in answering Research Question 3. The results of the GLM ANOVA and ANCOVA for each respective language are presented in Tables 49 thru 52.

Table 49 General Linear Model Procedure (ANOVA) TIMECOR (BASIC) (time to locate and correct logic errors in a BASIC program)					
Source of Variation	DF	TYPE I SS	Mean Square	F Value	Pr>F
TREATMENT	1	0.252820	0.252820	1.65	0.206
GEFTCD	2	0.510760	0.255380	1.67	0.202
TREATMENT*GEFTCD	2	0.044349	0.022174	0.14	0.865
PREREQ	1	0.768686	0.768686	5.01	0.031

\* p<.05

**Table 50**  
**General Linear Model Procedure (ANCOVA)**  
**TIMECOR (BASIC)**  
**(time to locate and correct logic errors in a BASIC program)**

Source of Variation	DF	TYPE III SS	Mean Square	F Value	Pr>F
TREATMENT	1	0.307438	0.307438	2.01	0.164
GEFTCD	2	0.136013	0.068006	0.44	0.645
TREATMENT*GEFTCD	2	0.006893	0.003446	0.02	0.977
PREREQ	1	0.768686	0.768686	5.01	0.031

\*  $p < .05$

**Table 51**  
**General Linear Model Procedure (ANOVA)**  
**TIMECOR (COBOL)**  
**(time to locate and correct logic errors in a COBOL program)**

Source of Variation	DF	TYPE I SS	Mean Square	F Value	Pr>F
TREATMENT	1	0.033566	0.033566	0.13	0.716
GEFTCD	2	1.115916	0.557958	2.23	0.123
TREATMENT*GEFTCD	2	0.440642	0.220032	0.88	0.424
PREREQ	1	3.704485	3.704485	14.83	0.000

\*  $p < .05$

**Table 52**  
**General Linear Model Procedure (ANCOVA)**  
**TIMECOR (COBOL)**  
**(time to locate and correct logic errors in a COBOL program)**

Source of Variation	DF	TYPE III SS	Mean Square	F Value	Pr>F
TREATMENT	1	0.286707	0.286707	1.15	0.291
GEFTCD	2	0.384108	0.192054	0.77	0.471
TREATMENT*GEFTCD	2	0.190779	0.095389	0.38	0.685
PREREQ	1	3.704482	3.704482	14.83	0.000

\*  $p < .05$

### Interpretations:

There was no statistically significant interaction of TREATMENT and GEFTCD (field dependence) on the dependent variable TIMECOR (the ability to locate and correct a program logic error) in either programming language (LANGUAGE). There were no significant main effect found for either independent variable, TREATMENT and GEFTCD, on the dependent variable TIMECOR.

After adjusting for the differences found in the programming prerequisite tests, there were no statistically significant differences found in the ability or time to locate a program logic error between the traditional and the interactive debugging groups. In addition, there were no statistically significant interactions found between the debugging treatment, field dependence and programming language.

### Secondary analysis

Each debugging posttest consisted of five different programs, each having a different debugging task. The complexity of the debugging task was designed to increase from program one to program five. Each of the five individual program test scores were analyzed in a 2 X 2 X 3 SAS General Linear Procedure (GLM-type) ANCOVA, homogeneity-of-slopes model. It was of interest to determine if a

particular type of programming debugging task was affected by the model.

Table 34 presents the results of a 2 x 2 x 3 ANCOVA, which analyzes the fourth program's test score based upon TREATMENT, LANGUAGE, and field dependence (GEFTCD) while adjusting for programming prerequisite skills.

Source of Variation	DF	TYPE III SS	Mean Square	F Value	Pr>F
LANGUAGE	1	17.8	17.8	0.06	0.811
TREATMENT	1	1238.8	1238.8	3.98	0.049*
LANGUAGE*TREATMENT	1	46.7	46.7	0.15	0.699
GEFTCD	2	228.5	114.28	0.37	0.694
LANGUAGE*GEFTCD	2	425.5	212.7	0.68	0.508
TREATMENT*GEFTCD	2	108.3	54.9	0.17	0.840
LANGUAGE*TREATMENT*GEFTCD	2	983.6	491.8	1.58	0.213
PREREQ	1	3360.3	3360.3	10.78	0.001

\*p<.05

#### Interpretations:

A statistically significant TREATMENT effect was found (p<.05) for the fourth program's test score. There were no other significant effects.

## SUMMARY AND CONCLUSIONS

### Summary

The purpose of this study was to evaluate the effects of computer program debugging tools, computer program languages, and field dependence on the ability of a student programmer to locate and correct logic errors in a computer program. Two intact groups of COBOL programming students and two intact groups of BASIC programming students participated in this study. One group was randomly assigned to the interactive debugging treatment and the other group was assigned to the traditional debugging treatment in each respective programming language.

Developing a computer program and debugging computer program errors are demanding problem solving tasks (Shneiderman, 1980). Locating and correcting a logic error in a computer program is probably the most difficult and time-consuming task in the program development process. Meyers (1979) reported that the process of locating and correcting a logic error in a computer program represented 95 percent of the total program development process time. Coupled with the annoying difficulties of admitting that one made a mistake, the frustrations of the program debugging

process may cause some students to become alienated from computer technology.

Debugging logic errors in a computer program involves a high-level heuristic that requires students to recognize the logic error, analyze the cause of the logic error, and to apply various syntactical and algorithmic skills to create a solution for the discovered error. This process requires significant cognitive resources. The use of the interactive computer program debugger was expected to: (a) relieve the constraints on short term memory, (b) permit students to view the execution steps of the program as written in the student's native program code, (c) provide various organizational cues that would enable students to structure the debugging process, and (d) provide an interactive control environment that would enable students to test various debugging strategies.

The interactive program debugger was expected to help field dependent students more than field independent students. Locating and correcting program logic errors requires the individual to take a critical program element out of the context of the program and to select the correct problem solving strategy to formulate a solution for the logic error. This process also requires students to understand the logic error in relation to the context of the program.



The trend of non-computer science majors attending computer programming classes and the diversity of cognitive styles has presented new challenges to computer programming curricula. The interactive computer program debugger may assist computer science educators in meeting those challenges.

#### Discussion of the Findings

The findings of this study can be subdivided into two sections: primary and secondary. The primary findings of this study included the results of three ANCOVA analyses that provided evidence to answer the three research questions. The secondary findings included various descriptive statistics, correlation analysis and other tests that may serve to clarify the purpose of the study or to serve as a basis for future research.

The results of the 2 x 2 x 3 ANCOVA (Table 48) indicated no significant effects after adjusting for differences found in prerequisite programming skills (PREREQ). These statistical findings provided important information that aided in answering Research Questions 1 and 2.

The evidence provided by this 2 x 2 x 3 Analysis of Covariance served as a basis for the following findings:

1. There was no statistically significant interaction found among the program debugging treatment, programming language, and field dependence.
2. There was no statistically significant interaction found between the program debugging treatment and programming language.
3. There was no statistically significant interaction found between the program debugging treatment and field dependence.
4. There was no significant interaction found between field dependence and programming language.
5. There was no statistically significant main effect found between the BASIC and COBOL programming languages in the ability to locate and correct a program logic error (LOCCOR).
6. There was no statistically significant main effect found among the field independent, field dependent, and indeterminate students in the ability to locate and correct a program logic error (LOCCOR).
7. There was no statistically significant main effect found between the traditional and interactive debugging treatment groups in the ability to locate and correct a program logic error (LOCCOR).

The statistical findings of two 2 x 3 ANCOVAs (see Tables 50 and 52) indicated that there are no significant differences in the time to locate and correct a logic error (TIMECOR) were found for any main effects or interaction when adjusted for differences found in prerequisite programming skills (PREREQ). These statistical findings provides important information that aided in answering Research Questions 2 and 3. The evidence provided by these 2 x 3 Analyses of Covariance served as a basis for the following findings:

1. There was no significant interaction found between the program debugging treatment and field dependence for either programming language.
2. There was no significant main effect found among field independent, field dependent, and indeterminate students and the time required to locate and correct a logic error in a computer program (TIMECOR) for either programming language.
3. There was no significant main effect found between the traditional and interactive debugging groups in the time required to locate and correct a logic error in a computer program (TIMECOR) for either programming language.

The lack of any significant main effects found between field dependence and the ability (LOCCOR) and time (TIMECOR) to locate and correct a program logic error did not concur with the results of previous research. As shown in Tables 47 thru 52, field independent students did not perform significantly better on the debugging posttest than field independent students.

Cavaiani (1989) investigated the influence of field dependence on the ability of a student programmer to locate and correct logic errors in a COBOL program. In the Cavaiani study, field dependent individuals did have significantly more difficulty in locating and correcting program logic errors. This study does not support Cavaiani's research.

However, since the lack of concurring results were found for both the interactive and traditional debugging treatments, these divergent findings cannot be fully explained by the use of the interactive program debugger. Other intervening variables may have contributed to these results, including the following: (a) the use of microcomputers in this study was different than the mainframe computers used in Cavaiani's study, (b) the program debugging tasks and the posttest administration conditions were different than Cavaiani's study, e.g., pen-and-pencil tests versus computer-based testing, and (c) the amount of instructional time and reinforcement practice devoted to program debugging strategies were different than

Cavaiani's study.

Cavaiani (1989) also studied the influence of different scoring schemes on the measurement of student's debugging skills. Though this study did use one of Cavaiani's scoring schemes in the debugging posttest, the interaction of the debugging tasks used on the posttest and the scoring scheme may also have contributed to the conflicting results.

Additional analyses of the data produced several other findings. These secondary findings include: (a) the importance of mastering program prerequisite skills in developing program debugging skills, (b) the practical size of the treatment effect of the interactive program debuggers, c) approximately 25% of the students participating in the interactive debugging treatment failed to use interactive debugging tools during the posttest, and d) a significant treatment effect was found when the posttest programs were analyzed individually.

First, as shown in Table 45, a significant correlation ( $r=.44$ ,  $p=.0001$ ) was found between the student's ability to locate and correct a program logic error (LOCCOR) and student's prerequisite test scores (PREREQ). In other words, mastery of programming prerequisite skills was significantly related to improved performance on the debugging posttest.

Second, while no significant statistical effects were found in this study, the absolute size of the observed effect between the interactive and traditional program

debugging treatments merits further discussion. As shown in Table 35, the interactive debugging mean test scores exceeded the traditional debugging total mean test scores (LOCCOR) by .38 standard deviations. The observed effect size was slightly larger for BASIC programming language groups than the COBOL programming language groups. Translated into percentage points, the effect was approximately 12.8%, or more than one letter grade difference.

Locating and correcting logic errors is perhaps the most demanding and frustrating task for a student programmer. Any improvement in this instructional area that provides an increase in student performance by more than one letter grade deserves serious consideration as a instructional tool in the computer programming curriculum. Yet, the nonsignificant results indicate that the .38 standard deviations is attributable to sampling error.

Third, approximately 25% of the students who had participated in the interactive debugging sections reported no use of any interactive debugging tool during the debugging posttest (see Tables 41 and 43). The students who participated in the interactive debugging sections could have chosen from one of three interactive debugging tools. "Stepping" is a tool that permits the student to view the text of their native program code as their program executes. This was the most frequently used interactive debugging

tool. "Querying Variables," which permitted students to inspect the contents of variables during program execution, was the second most frequently used interactive debugging tool. Setting "Break Points," which permitted students to toggle between normal and interactive debugging execution modes, was rarely used by interactive debugging student programmers during the posttest.

Students who did use the interactive debugging tools reported that their perceived value of the tools increased as the posttest programs became more challenging, i.e., programs four and five (see Tables 41 and 43). This trend was especially significant for SECT4, the COBOL interactive debugging section. A significant correlation for SECT4 was found between the student's reported value of the interactive debugging tool and the test score for programs four and five.

These reported results seem to indicate that students who participated in the interactive debugging sections were more likely to use the interactive debugging tools as the debugging task became more challenging. While the validity and the interpretation of this self-reported data can be questioned, the pattern of usage was confirmed by the observations of the test administrators.

Fourth, while there was no statistically significant main effects or interactions found when the total posttest debugging score (LOCCOR) was used as the dependent variable,

a significant TREATMENT effect was found for the fourth program's test scores (see Table 53). Each debugging posttest consisted of five programs with each having a different debugging task. The complexity of the debugging task was designed to increase with each program, from program one to program five.

Data from Table 39 provides evidence that programs four and five had the largest discrimination indexes (.92 and .96) and the largest point biserial correlation coefficients (.84 and .75). This evidence suggests that value of the interactive debugger may be more applicable to correcting logic errors in advanced level algorithms found in more advanced level programming courses than those typically found in entry level programming courses.

### Conclusions

Based upon the analysis of data and previous discussion of the findings, the following conclusions can be made:

- 1) The use of either the interactive program debugging tool or the traditional program debugging tool does not effect an entry-level programmer's ability to locate and correct a program logic error.



- 2) **The use of either the interactive program debugging tool or the traditional program debugging tool does not effect an entry-level programmer's time to locate and correct a program logic error. If the instructional resources for interactive debugging are not available or instructional time is limited, instructional use of traditional program debugging tools is adequate.**
  
- 3) **The use of either the BASIC or the COBOL programming language does not effect an entry-level programmer's ability to locate and correct a program logic error.**
  
- 4) **The use of either the BASIC or the COBOL programming language does not effect an entry-level programmer's time to locate and correct a program logic error. BASIC was designed to be an easy-to-learn programming language for student programmers. COBOL, in spite of its wide use in business, has been considered a verbose programming language that is difficult for students to learn. Considering the relative performance of the programming debugging posttest**

and the three prerequisite tests, there is no evidence to support the assumption that COBOL is a more difficult to learn educational programming language. The choice of a programming language in the curriculum should consider other factors, e.g., availability, cost, use, and the importance of providing career opportunities.

- 5) **Field dependence does not effect an entry-level programmer's ability to locate and correct a program logic error.**
  
- 6) **Field dependence does not effect an entry-level programmer's time to locate and correct a program logic error. It was expected that field dependent students would have difficulty in understanding the logic error in relation to the context of the program. This result was not found. Perhaps altering the instructional presentation or content may provide opportunities for field dependent programmers to learn to correct program logic errors.**

- 7) **Field dependence does not interact with the programming language or program debugging tool in an entry-level programmer's ability to locate and correct a program logic error.**
- 8) **Field dependence does not interact with the choice of programming language or program debugging method in an entry-level programmer's time to locate and correct a program logic error.**
- 9) **Programming language does not interact with the choice of program debugging tool in an entry-level programmer's ability and time to locate and correct a program logic error.**
- 10) **Mastering computer programming prerequisite skills is significantly related to an entry-level programmer's ability and time to locate and correct a program logic error. This study has provided evidence that reinforces the conventional wisdom of some educators, that technology is not a substitute for learning fundamentals. Again, the theory of Bloom's taxonomy has been reaffirmed: without acquiring prerequisite knowledge and skills a student cannot apply those skills.**

- 11) **Students tend to rely on traditional program debugging methods and will only use the interactive debugging tools when they perceive the debugging task to be challenging or unfamiliar.**
  
- 12) **When using the interactive debugging method, students tend to rely on the "Stepping" tool and fail to take advantage of more advance interactive debugging options, e.g., "Setting Break Points" or "Watching Variables."**
  
- 13) **Full benefit of the interactive debugger may be realized when applied to advanced level programming curriculum and unfamiliar algorithmic tasks, rather than those encountered by entry-level college programming students.**
  
- 14) **No demographic or background variables were significantly related to an entry-level student programmer's ability and time to locate and correct an program logic error. Experience in computer programming, wordprocessing, or the ability to prepare spreadsheets was not significantly related to program debugging abilities. In addition, the availability of a home computer or the use of a computer at work was not**

significantly related to program debugging abilities. While owning a home computer may offer students convenience, it has no relationship to the development of students' program debugging abilities.

#### Further Research Recommendations

This study has only "opened the door" for investigating the process of student learning with respect to debugging logic errors in a computer program. While studies by Pressman (1987) and Shneiderman (1980) have documented the frustrations of the program debugging process, this study was an attempt to identify instructional methods to improve the process.

Before detailed recommendations for further study can be discussed, the importance of the size of the large error term as shown in Table 46 needs to be addressed. The error term represents the sources of variation that are not attributable to the effects of the independent variables and covariate. The use of intact groups does not allow the researcher the ability to control or manipulate all relevant variables, which increases the chance that confounding variables may be contributing to the error term.

The findings of this study suggest that the effect size of the interactive computer debugger group was large enough

to be of practical instructional value, yet no significant statistical differences were found. The size of the sample and the size of the effect were sufficient to provide reasonable statistical power. However, the size of the error term relative to practical significance of the effect requires further explanation.

The results of this study indicate several specific needs for future research. These include:

- 1) In order to more precisely define the results of this study, the use of the program debugging tool during the program debugging posttest needs to be better controlled.
  
- 2) The relationship between the program debugging task and the debugging tool needs to be studied further. In addition, the similarity of the debugging tasks of the posttest and the instructional content of the course requires further study.

The program debugging test should be modified to include more advanced debugging tasks or those unfamiliar to the student. A significant effect for the interactive program debugger was found for

the fourth program of the program debugging posttest. Program four contained a challenging and unfamiliar program debugging task. This evidence suggests that either the debugging posttest instrument used in this study might not have been sufficiently difficult to measure the effects of the interactive program debugger or that the effectiveness of the interactive program debugger may be only found in either advanced computer science curricula or the professional programming environment.

- 3) **The instructional presentation of program debugging skills within the sequence of the computer programming curricula needs to be studied further.**

There is considerable debate among computer programming educators concerning the appropriate scheduling of program debugging skills within the sequence of computer programming curricula. It has been argued by some computer science faculty members that scheduling the presentation of program debugging skills early in the semester, while students were simultaneously mastering prerequisite programming skills, may overwhelm

them. In consideration of these concerns, the program debugging treatment of this study was scheduled later in the semester.

Presenting a new program debugging strategy at the end of an instructional period may not overcome poor program debugging habits learned by students earlier in the semester. Tables 42 and 43 presented evidence that many students in the interactive debugging treatment groups did not use the interactive program debugger tools during the posttest, but rather relied on other program debugging strategies, i.e., reviewing program source code and outputs. Since these more traditional debugging methods were learned earlier in the semester, these previously learned debugging habits may have been too difficult to alter.

Because 25% of the students never used any interactive debugging tools during the posttest (see Tables 42 and 43), it is difficult to measure the effectiveness of the interactive debugging tool. The lack of use of the interactive program debugger may provide a partial explanation for large size of the error term found in the ANCOVA analysis. If all students in the interactive group had used the interactive debugging tools during



the posttest, the study may have found a significant advantage for the interactive debugging treatment. There is a need to investigate the appropriate scheduling of program debugging skills within the sequence of computer programming curricula. In addition, there is also a need to study student attitudes toward the various program debugging strategies.

- 4) **It is recommended that further research should be conducted to determine the reasons for the lack of student's use of the interactive program debugger.**

The lack of use of the interactive program debugger during the debugging posttest may indicate an inadequacy of the interactive debugging tool itself. For example, the interactive debugging tools may have been too difficult or students may have perceived the power of the debugging tool as too limited for the debugging tasks presented.

The MicroFocus COBOL interactive debugger was more easier to use and more powerful than the MicroSoft QuickBASIC interactive debugger (see Tables 1 thru 4). However, both interactive program debuggers had some disadvantages when

compared to the process of reviewing printed program code. Printed program code typically displayed more lines of the program source code as compared to the interactive program debugger's display on the computer screen (sixty lines as compared to twenty lines). In addition, students could write debugging strategies directly on their paper documentation, but were restricted from writing notes on the computer screen.

- 5) **Future research should address the reasons why field dependent individuals did not perform significantly different than field independent individuals in debugging program logic errors as found in previous research.**

The relationship between the field dependence and the sequence of presentation, the amount of instructional time, and the type of instructional materials used in this study should be researched further.

- 6) **Additional study is recommended to investigate the relationship between the cognitive style field dependence and the complexity and similarity of the debugging task.**

- 7) **Further research into program debugging skills and tools and their relation to different populations, e.g., computer science majors and adult learners, and other programming languages is recommended.**

The participants of this study were predominately noncomputer science majors. This study should be repeated for computer science majors enrolled in advanced level computer programming courses. Furthermore, research into the program debugging skills and tools used by professional programmers may decrease the program development time and provide cost savings for business, industry and government.

- 9) **The relationship between unstructured, structured and object-oriented program development paradigms and program debugging skills and tools need to be studied further.**

In addition to the previous recommendations, it is recommended that further research be conducted in order to more fully understand the cognitive processes of debugging a computer program and their relationship between program debugging tools, cognitive style, programming languages, and other academic disciplines. The following recommendations

for further research into other aspects of program debugging are made:

- 1) **Future research is needed to more fully understand the cognitive processes used in program debugging and their relationship to program debugging tools.**

For example, research into the cognitive processes that enable programmers to perceive the location or cause of logic error is needed. The role of color, graphical representations, e.g., structured chart animation, type font and other cues in program debugging should be considered in future research.

Future research is needed concerning the cognitive processes used by computer programmers. In addition, and debugging tools that serve to organize and supplement constrained short term memory resources should be studied. For example, structured and object-oriented programming strategies may improve program debugging skills through the application of various "chunking" strategies. Another important need for future research involves the investigation of the cognitive strategies used to encode, store, organize and retrieve debugging knowledge and

skills stored in long term memory.

Dijkstra et al. (1989) has suggested that the automation of computer science curriculum may create a situation where students may no longer have any concrete understanding of the actual processes of writing programs. One may argue that the automation of the debugging process may not effectively engage the programmer in the problem solving process and may actually inhibit long term storage of acquired knowledge. This argument may have some merit.

Craik and Lockhart (1972) suggested that the greater the level and depth of processing, the greater the possibility that information will be encoded and stored in long term memory. If automation of the program debugging process fails to engage the student in elaborative processes, then the interactive program debugger may not contribute to a significant meaningful, learning experience.

- 2) **Additional research into the relationship between problem solving strategies, e.g. backtracking, top-down, representative, divide and conquer, and programming debugging skills and tools is needed.**

- 3) The ability to transfer interactive program debugging skills to other computer program debugging environments needs to be studied.**

Swaine (1990) was concerned whether debugging tools would be abused by poor programmers. One must consider the possibility that problem solving skills developed through student use of interactive program debuggers may not transfer to other programming situations in which the interactive program debugger is not used.

The arguments against interactive program debuggers are similar to the ones proposed regarding the use of calculators in schools. Opponents argued that calculator usage should not become a substitute for fundamental computational skills. Similarly, one must consider whether the specific programming debugging skills developed through the use of the interactive program debugger tool will laterally transfer to other program debugging skills.

- 4) The ability to transfer interactive program debugging skills to other disciplines needs to be studied.**

Some educators have viewed computer programming as a tool to help students develop problem solving skills (Martin & Hearne, 1990; McCoy & Dodl, 1989) that may provide benefits to other disciplines. Research is needed to determine if problem solving skills developed in computer programming curricula will transfer to other disciplines. The role of alternative debugging tools and their relationship to developing problem solving abilities in other disciplines may also need to be addressed.

- 5) **Research could be conducted to determine the relationship between program debugging skills and tools and other cognitive style constructs.**

For example, the concept of "leveling" represents the internal cognitive processes of individuals who merge perceived objects or events with similar but not identical objects and events recalled from memory. On the other hand, the concept of "sharpening" represents the internal cognitive processes of individuals who are less likely to confuse similar objects and would be more likely to magnify small differences between similar information stored in memory. "Levelers" may be expected to experience some difficulties in

learning program debugging skills; however, once learned, they may be able to apply these skills to dissimilar and more challenging debugging tasks.

The cognitive processing continuum of "reflectivity" versus "impulsivity" also may be an important cognitive style to consider in relationship to the interactive programmer debugger. The use of the interactive program debugger may benefit "impulsive" individuals by increasing the accuracy of their program debugging skills. On the other hand, the use of the interactive program debugger may benefit "reflective" individuals by decreasing the amount of time to debug a program.

Other cognitive style constructs that should be considered in future research include: converging versus diverging, scanning, conceptual integration, and conceptual discrimination.



## **BIBLIOGRAPHY**

**BIBLIOGRAPHY**

- Anderson, J.R. (1985). Cognitive psychology and its implications (2nd ed.). New York, NY: Freeman.
- Arnone, M. P., & Grabowski, B.L. (1991). Effects of variations in learner control on children's curiosity and learning from interactive video. In M.R. Simonson & C. Hargrave (Eds.), 13th Proceedings of Selected Research Presentations of the Association for Education Communications and Technology. Ames, Iowa: Iowa State University. 45-67.
- Bell, D., Morrey & Pugh, J. (1987) Software engineering a programming approach. Englewood Cliffs, NJ: Prentice-Hall.
- Benander, A. C. & Benander, B. A. (Summer, 1991). An analysis of debugging techniques. Journal of Research on Computing and Education, 37(2), 447-455.

- Bower, G. H. (1972). Mental imagery and associative learning. In L.W. Gregg (Ed.), Cognition in learning and memory. New York: Wiley.
- Carrier, C. A., & Jonassen, D. A. (1988). Adapting courseware to accommodate individual differences. In D. A. Jonassen (Ed.), Instructional Designs for Microcomputer Software. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Carver, Doris L. (1989). Programmer variations in software debugging. Communications of the ACM, 32(12), 1100-1112.
- Cavaiani, Thomas P. (Summer, 1989). Cognitive style and diagnostic skills of student programmers. Journal on Research on Computing in Education, 37(2), 411-421.
- Cleborne, D. M. (1989, July). The harmful effects of excessive optimism in educational computing. Educational Technology, 23-29.
- Clements, D.H. & Gullo, D.F. (1985). Effects of computer programming on young children's cognition. Journal of Educational Psychology, 76(6), 1051-1058.

- Cohen, J. (1969). Statistical analysis for the behavioral sciences. New York: Academic Press.
- Cohen, R. (1969, October). Conceptual styles, cultural conflict and nonverbal tests of intelligence. American Anthropologist, 828-859.
- Conrad, R. (1971). The chronology of the development of covert speech in children. British Journal of Psychology, 5, 398-405.
- Conrad, R. (1972). Short term memory in deaf: A test for speech encoding. British Journal of Psychology, 55, 429-432.
- Craik, F.I.M. & Lockart, R.S. (1974). Levels of processing: A framework for memory research. Journal of Verbal Learning and Verbal Behavior, 11, 671-684.
- Cromier, S. M. & Hagman J.D. (1987). Introduction, In S.M. Cromier & J.D. Hagman (Eds.), Transfer of Learning: Contemporary Research and Applications, San Diego, CA: Academic Press.
- Cronbach, L.J., & Snow, R.E. Aptitudes and instructional methods. New York: Irvington Press, 1977.

- Davis, W. A. (1983). Operating systems: A systematic view. Menlo Park, CA: Addison-Wesley Publishing Company.
- Dijkstra, P. J. et al., (1989). A debate on teaching computer science, Communications of the ACM, 32(12), 1397-1416.
- French, M. (1983). A supplantation approach to the design of instructional visuals. Paper presented at the National Convention of the Association of Communications and Technology, New Orleans, LO, January, 1983.
- Gick, M.L. & Holyoak, K.J. (1987). The cognitive basis of knowledge transfer, In S.M. Cromier & J.D. Hagman (Eds.), Transfer of learning: Contemporary research and applications, San Diego, CA: Academic Press.
- Gilbert, P. (1983). Software design and development. Chicago, IL: Science Research Associates.
- Goetzfried, L. & Hannafin, M.J. (1985). The effects of the locus of CAI control strategies on the learning of mathematics rules. American Education Research Journal, 22(2), 273-278.

- Gray, W. D. & Orasanu, J. M. (1987). The transfer of learning, In S.M. Cromier & J.D. Hagman (Eds.) Transfer of Learning: Contemporary Research and Applications. San Diego, CA: Academic Press.
- Hannafin, M. J. (1984). Guidelines for using locus of instructional control in the design of computer assisted instruction. Journal of Instructional Development, 7(3), 6-10.
- Hicks, C. (1973). Fundamental concepts in the design of experiments (2nd ed.). New York: Holt, Reinhart and Winston.
- Isaac, S. & Michael, W. (1981). Handbook in research and evaluation (2nd ed.). San Deigo, CA: EdITS Publishers.
- Jesky, R. & Berry, L. (1991). The effects of pictorial complexity and cognitve style on visual recall memory. In M.R. Simonson & C. Hargrave (Eds.), 13th Proceedings of Selected Research Presentations of the Association for Education Communications and Technology. Ames, Iowa: Iowa State University. 290-296.

- Jonassen, D. A. & Tonnysen, R. D. (1983). Effect of adaptive advisement of perception in learner-controlled, computer-based instruction, using a rule-learning task. Education and Communication Technology Journal, 31(4), 226-236.
- Jonassen, D. A. (1988). Learning strategies in courseware. In D.A. Jonassen, (Ed.), Instructional Designs for Microcomputer Software, Hillsdale, NJ: Lawrence Erlbaum Associates.
- Kirk, R. A. (1968). Experimental design: Procedures for the behavioral sciences. Belmont, CA: Brooks/Cole Publishing Company.
- Koohang, Alex, (1989). A study of attitudes toward computers: Anxiety, confidence, liking, and the perception of usefulness. Journal of Research on Computing and Education, 37(4), 137-150.
- Krendle, K. A., & Liberman, B. A. (1988). Computers and learning: A review of recent research. Journal of Educational Computing Research, 4(4), 367-389.
- Kulhavy, R.W. & Swenson, I. (1975). Imagery instructions for the comprehension of text. British Journal of Educational Psychology, 45(2), 47-51.

- Kulhavy, R. W. Schwartz, N.H. & Shaha, S.H. (1983).  
Spatial representations of maps. American Journal of Psychology, 96, 337-351.
- Kutscher, R. (1990). Outlook 2000: The major trends,  
(Occupational Outlook Quarterly), Washington, DC:  
Department of Labor, 2-7.
- Laverty, J. (1990). [Survey of instructional strategies  
in higher education computer courses]. Unpublished  
raw data.
- Lee, M. J. (1991). Metacognitive and cognitive effects  
of different loci of instructional control. In M.R.  
Simonson & C. Hargrave (Eds.), 13th Proceedings of  
Selected Research Presentations of the Association  
for Education Communications and Technology. Ames,  
Iowa: Iowa State University. 433-459.
- Lepper, M.R. (1985). Micro computers in education:  
Motivational and social issues. American  
Psychologist, 40(1), 1-18.
- Levin, J.R. (Summer, 1975). Determining sample size  
for planned and post hoc analysis of variance  
comparisons. Journal of Educational Measurement,  
12(2), 99-108.



- Linn, M.C. (1985). The cognitive consequences of programming instruction in the classroom. Educational Researcher, 14(5), 14-29.
- Linn, M.C. & Kyllonen, P. (1981). The field dependence-independence construct: some, one, or none. Journal of Educational Psychology, 73, 261-273.
- Maddux, C. (February, 1989). Logo: Scientific dedication or religious fanaticism in the 1990's?, Educational Technology, 22-26.
- Maddux, C. (July, 1989). The harmful effects of excessive optimism in educational computing. Educational Technology, 23-29.
- Maier, N.R.F. & Jansen, J.C. (1969). Are good problem solvers also creative? Psychological Reports, 24, 139-146.
- Mattoon, J. & Klein, J. & Thurman, R. (1991). Learner control versus computer control in instruction simulation. In M.R. Simonson & C. Hargrave (Eds.), 13th Proceedings of Selected Research Presentations of the Association for Education Communications and Technology, Ames, Iowa: Iowa State University, 481-497.

- Maxwell, S. E., & Delaney, H.D. (1990). Designing and analyzing data: A model comparison prespective. Belmont, CA: Wadsworth.
- McCoy, L.P. & Dodi, N.R. Dodi. (1989). Computer programming experience and mathematical problem solving. Journal of Research on Computing and Education, 37(3), 14-25.
- Meyer, G. (1979). The art of software testing. New York: Wiley.
- Miller, G.A. (1956). The magical number seven, plus or minus two. Psychological Review, 63, 81-97.
- Newell, A. & Simon, H. (1972). Human problem solving. Englewood Cliffs, NJ: Prentice-Hall.
- Nickerson, R.S. (1982). Computer programming as a vehicle for teaching skills. Thinking: The Journal of Philosophy for Children, 4, 42-48.
- Nickerson, Robert C. (1986). Fundamentals of Structured Programming (2nd ed.). Glenville, IL: Scott, Foresman and Company.
- Obrien, J. (1985). Computers in Business Management. Homewood IL: Richard D. Irwin, Inc.

- Ormrod, J.E. (1990). Human Learning: Theories, Principles, and Educational Applications. Columbus, OH: Merrill.
- Osgood, C. E. (1949). The similarity paradox in human learning: A resolution. Psychological Review, 56, 132-143.
- Occupational Outlook Handbook (1988). U.S. Department of Labor, Bureau of Statistics, 66-67, 203-234.
- Pappert, S. (1980). Mindstorms: Computers, children and powerful ideas. New York: Basic Books.
- Pascaul-Leone, J., Ammon, P., Goodman, D., & Subleman, I. (1978). Piagetian theory and neo-Piagetian analysis as psychological guides in education. In J. M. Gallagher & J. Easley (Eds.), Knowledge and Development Vol.2: Piaget and Education. New York: Plenum Press.
- Peterson, L.R. & Peterson, M.J. (1959). Short term retention of individual items. Journal of Experimental Psychology, 58, 193-198.
- Picthrone, B. (1983). A missing link between cognitive science and classroom practice. Information Science, 11, (4), 218-312.

- Pierson, Joan K. & Horn, Jeretts A. (1984).  
Syntactical errors in programs written by students  
in business programming. AEDS Journal, 17(3),  
53-60.
- Pressman, Rodger, S. (1987). Software Engineering: A  
Practitioner's Approach (2nd ed.), New York, McGraw-  
Hill, Inc.
- Q&A User's Manual (1988). Symantec Corporation,  
Cupertine, CA
- Rameriez M. & Castenda, A. (1979). Cultural  
Democracy, Bicognitive Development and Education.  
New York, NY: Academic Press.
- Rieber, L.P. (1991). "The effects of visual grouping on  
learning from computer animated presentations,  
in M.R. Simonson & C. Hargrave (Eds.),  
13th Proceedings of Selected Research Presentations  
of the Association for Education Communications and  
Technology, Ames, Iowa: Iowa State University.  
681-690.
- Reitman, W.R. (1965). Cognition and thought. New York:  
Wiley.

- Resnick, L.B. & Glasser, R. (1976). Problem solving and intelligence. In L.B. Resnick (Ed.), The Nature of Intelligence. Hillsdale, NJ: Erlbaum.
- Restle, F. & Davis, J.H. (1962). Success and the speed of problem solving by individuals and groups. Psychological Review, 69, 520-536.
- Shepard, R.N. & Metzler, J. (1971). Mental rotation of three dimensional objects. Science, 171, 701-703.
- Shneiderman, B. (1980). Software Psychology. New York: Winthrop Publishers.
- Simon, H.A. (1974). How Big is a Chunk? Science, 183, 482-488.
- Steinberg, E.R. (1977). Review of student control in computer-assisted instruction. Journal of Computer Based Instruction, 3(3), 84-90.
- Stern, N. & Stern, R., 1990. Structured COBOL Programming (6th ed.). New York, NY: John Wiley & Sons.
- Swaine, M. (1990). Wrapping up software development '90. Dr. Dobb's Journal, 15(4), 119-123.
- Tenner, E. The computer and higher education: A new definition of education? Change, 16(3), 22-27.

Tobias, S. (1976). Achievement treatment interaction.

Review of Educational Research , 46(1), 61-74.

Violato, C., Marini, A. & Hunter, W. (1989). A

confirmatory factor analysis of a four factored model of attitudes towards computers: A study of preservice teachers. Journal of Research on Computing in Education, 37(1), 199-213.

Ward, R. (1988). Beyond design: The discipline of debugging. Computer Language, 37-38.

Winfred, A. & Hart, D., (1990). Empirical relationships between cognitive ability and computer familiarity.

Journal of Research on Computing in Education , 38(2), 457-463.

Witkin, H. A. (1949). Perception of body position of the visual field . Psychological Monograph, 63, 1-46.

Witkin, H. & Goodenough, D. (1981). Cognitive style: Essence and origins. New York: International Universities Press, Inc.

Witkin, H., Goodenough, D. & Oltman P. (1979).

Psychological differentiation: Cultural status. Journal of Personality and Social Psychology. 37(7), 1127-1145.

Witkin, H., Oltman, P. Raskin, E. & Karp, S.

(1971). A manual for the embedded figures tests.

Palo Alto, Ca: Consulting Psychologists Press, Inc.

Yourdan, E. & Constantine, L. (1979). Structured

design. Englewood Cliffs, NJ: Prentice-Hall.

## **APPENDICES**



## APPENDIX A

### Prerequisite COBOL Knowledge

Since computer programming debugging is a high level educational objective, certain low level educational objectives must first be achieved. Students must learn the how to develop, edit, code and compile a program. Construction of program flow charts, structured charts and pseudocode are considered important program development skills (Fiengol & Wolf, 1988; Nickerson, 1987; Stern & Stern, 1988). Knowledge of how to use a program editor and COBOL syntax is an important prerequisite. Before a student programmer can debug logic errors in a COBOL, they must be able to code, compile the program and correct any syntax errors (Nickerson, 1987; Stern & Stern, 1988).

The specific prerequisite skills required to debug logic errors in a COBOL program are:

1. the program development life cycle,
2. the proper use a program development tool to plan the coding of a program, i.e., flowcharting or structure charts,

3. the proper use of SPFPC, a microcomputer program editor,
4. the proper procedures to translate and execute a COBOL program using MicroFocus Work Bench, a COBOL program development tool,
5. the basic coding requirements to write a COBOL program, which includes syntax of the four divisions of the source program that will input and output a sequential data file,
6. the syntax requirements to perform the three control structures: (a) sequence, (b) iteration and, (c) selection.
7. the syntax and algorithmic requirements to perform total accumulation, control paging, conditional calculations and high/low values.
8. the ability to distinguish between a syntax error, execution error and a logic error.

## APPENDIX B

### Prerequisite BASIC Knowledge

Since computer programming debugging is a high level educational objective, certain low level educational objectives must first be achieved. Students must learn the how to develop, edit, code and compile a program. Construction of program flow charts, structured charts and pseudocode are considered important program development skills (Fiengol & Wolf, 1988; Nickerson, 1987; Stern & Stern, 1988). Knowledge of how to use a Quick Basic editor and BASIC syntax is an important prerequisite. Before a student programmer can debug logic errors in a BASIC, they must be able to code, compile the program and correct any syntax errors (Nickerson, 1987; Stern & Stern, 1988).

The specific prerequisite skills required to debug logic errors in a BASIC program are:

1. the program development life cycle,
2. the proper use a program development tool to plan the coding of a program, i.e., flowcharting or structure charts,
3. the proper use of a Quick Basic editor,
4. the proper procedures to save, load and execute a

BASIC program using QuickBASIC,

5. the basic coding requirements to write a COBOL program, which includes the syntax of the source program that will input and output a sequential file.
6. the syntax requirements for the three control structures: sequence, iteration and selection.
7. the syntax and algorithmic requirements to perform total accumulation, control paging, conditional calculations and high/low values,
8. the ability to distinguish between a syntax error, execution error and a logic error,
9. the ability to create and use subprograms.

## APPENDIX C

### Program Syntax and Execution Errors

A syntax error is "an error caused by the violation of a programming rule" (Stern & Stern, 1991, p.751). Syntax errors are encountered by programmers when a program is translated from source code into executable binary code. All computer programs must be first translated into computer understandable code, before it can be executed. Typical syntax errors include: (a) misspelling a programming statement, (b) incorrect usage of the language punctuation (called delimiters), (c) incorrect usage of a data type, or (d) illegal parameters.

An execution error "will prevent program execution. These errors will result in a program interrupt" (Stern & Stern, 1991, p. 381). Execution errors will be discovered after a program is translated and begins to execute. At some point during a program's execution, the execution error will cause the program to stop executing. Typical execution errors include: (a) Inter procedural calls do not match in number or type, (b) unable to access an external data file, (c) data file not appropriately opened, (d) data field overflow, or (e) divide by zero.

**APPENDIX D****CI201 BUSINESS PROGRAMMING      JOSEPH P.(PACKY) LAVERTY****Class Days: Monday, Wednesday and Friday****Office Hours: To be Announced****PREREQUISITE COURSES: AC101 Accounting Principles I and  
CI101 Introduction To Computers  
or Equivalent****COBOL PREREQUISITE STUDENT ABILITIES:**

- 1) Students should be able to identify and explain the major hardware and software components of a microcomputer.
- 2) Students should be able to understand and execute MSDOS operating system commands, i.e., FORMAT, DIR, DISKCOPY, CLS, PRINT, etc.
- 3) Students should be able to LOGIN to the Novell microcomputer network system, use the student menu system. Students should be able to understand the concepts of printing on a Novell network.

**COBOL COURSE DESCRIPTION:**

An introduction to structured COBOL and programming techniques. Logical structure, modular design and documentation techniques are presented. The student will become familiar with the syntax and logic of COBOL by applying the language to a sequence of increasingly complex business applications. Processing techniques for one-level

tables are discussed, and the fundamental elements of sequential file processing are presented.

**COBOL PROGRAMMING COURSE OBJECTIVES:**

The fundamental objective of Business Programming is to introduce the student to the fundamental characteristics and the application of programming to business through the COBOL programming language. The important aspects to be stressed in this class include:

- 1) Students should be able to apply the concepts of the program development life cycle to solve various business problems.
- 2) Students should be able to develop a flowchart that will graphically illustrate a solution to a business or scientific problem.
- 3) Students should be able to understand the syntax of the COBOL programming language. Selected syntax topics include: the Four Divisions of a COBOL program, Sequential file processing (OPEN..READ..WRITE..CLOSE), iteration (PERFORM ..UNTIL), Data computation (ADD, SUBTRACT, etc.), logical selection (IF...ELSE, AND, OR), and single level arrays.
- 4) Students should be able to write various COBOL that will perform a required business task.

- 5) Students should be able to identify and correct various syntax, execution and logic errors in the development of their COBOL programs.
- 6) Students should understand and apply the concepts of structured programming style in the development of their COBOL programs.
- 7) Students will be encouraged to professionally prepare all assignments and to meet assignment deadlines in a timely fashion.

**TEXTBOOK:** "Structured COBOL Programming", (1991), Stern, R. A., Stern, N., New York, New York: John Wiley & Sons

**COURSE MATERIAL:** Flowcharting Template, Diskettes: two 3 1/2 inch high density (if you plan to code your program at school) or, two 5 1/4 inch double density.

**TEACHING PROCEDURES:**

The methods used in this course include lecture accompanied by instructional handouts, illustration of program examples on an overhead computer projection device, hands-on exercises and program assignments to reinforce the concepts and applications, and use of the textbook.

<b>GRADE ALLOCATION POLICY</b>		<b>points</b>
3 objective tests @ 150 points or 45%		450
2 written assignments @ 50 points or 10%		100
4 programs (points to be assigned)		300
final examination (comprehensive review)		150
Total		1000

1000-900    A    899-800    B    799-700    C    699-600    D  
 Below 600    F



**APPENDIX E****CS4/007 BASIC PROGRAMMING****JOSEPH P. (PACKY) LAVERTY****Class Days: Tuesday and Thursday****Office Hours: To be announced****BASIC PROGRAMMING COURSE DESCRIPTION**

This is the first course in computer science. It is designed to be of special interest to students majoring in one of the social sciences or humanities.

**BASIC PROGRAMMING COURSE OBJECTIVES:**

- 1) Students should be able to identify and explain the major hardware and software components of a microcomputer.
- 2) Students should be able to understand and execute MSDOS operating system commands, i.e., FORMAT, DIR, DISKCOPY, CLS, PRINT, etc.
- 3) Students should be able to LOGIN to the Novell microcomputer network system using the student menu system. Students should be able to understand the concepts of printing on a Novell network.

- 4) Students should be able to apply the concepts of the program development life cycle to solve various business and scientific problems.
- 5) Students should be able to understand the syntax of the QuickBASIC programming language. Selected syntax topics include: interactive processing (INPUT), logical selection (IF...THEN), iteration (DO...WHILE), mathematical functions, string manipulation, sequential file processing and arrays.
- 6) Students should be able to develop a flowchart that will graphically illustrate a solution to a business or scientific problem.
- 7) Students should be able to write various Quick Basic programs that will perform a required business or scientific task.
- 8) Students should be able to identify and correct various syntax, execution and logic errors in the development of their Quick Basic programs.
- 9) Students should understand and apply the concepts of structured programming style in the development of their Quick Basic programs.

- 10) Students will be encouraged to professionally prepare all assignments and to meet assignment deadlines in a timely fashion.

**TEXTBOOK:** "MICROSOFT QUICKBASIC: An Introduction to Structured Programming", (1991), Schneider, David, New Jersey: Dellen Publishing Company, a division of MacMillan Publishing Company.

**COURSE MATERIAL:** Flowcharting Template, Diskettes: two 3 1/2 inch high density (if you plan to code your program at school) or, two 5 1/4 inch double density.

**TEACHING PROCEDURES:**

The methods used in this course include lecture accompanied by instructional handouts, illustration of program examples on an overhead computer projection device, hands-on exercises and program assignments to reinforce the concepts and applications, and use of the textbook.

**GRADE ALLOCATION POLICY**

	points
3 objective tests @ 150 points or 45%	450
2 written assignments @ 50 points or 10%	100
6 programs @ 50 points or 30%	300
final examination (comprehensive review)	150
Total	1000
1000-900    A    899-800    B    799-700    C    699-600    D	
Below 600    F	

## APPENDIX F

## PRE-COURSE STUDENT SURVEY

NAME \_\_\_\_\_ SEMESTER \_\_\_\_\_  
 AGE \_\_\_\_\_ SEX (M) \_\_\_ (F) \_\_\_ MAJOR \_\_\_\_\_  
 MINOR \_\_\_\_\_

- Total number of college credits earned to date \_\_\_\_\_.  
(not including this semester)
- List any computer/programming language courses that you have completed.

Course Name	Computer Type Used	Programming Language Used
_____	_____	_____
_____	_____	_____
_____	_____	_____

- List the types of computers that you have used.  
(You may write multiple answers.)

High School	College	Work
_____	_____	_____
_____	_____	_____

- How long have you used a microcomputer?  
(Approx.) Months \_\_\_\_ Years \_\_\_\_
- Name the microcomputer software packages (ex. Lotus 1-2-3) that you have used.
- List any formal training courses on microcomputer hardware or software that you may have attended (ex. Lotus 1-2-3), company courses for a day?).

7. Check only **ONE** of the following that best describes your computer availability at home?

I have no computer at home. \_\_\_\_\_

I have used my computer at home:

- A. Mostly for pleasure, i.e., video games. \_\_\_\_\_
- B. Mostly for word processing and spread sheet homework assignments. \_\_\_\_\_
- C. Mostly for writing computer programs. \_\_\_\_\_
- D. Business \_\_\_\_\_
- E. Telecommunication or networking. \_\_\_\_\_
- F. Any other - please list: \_\_\_\_\_  
\_\_\_\_\_

8. Are you presently working? \_\_\_\_\_ Part-time \_\_\_\_\_ Hours per week  
\_\_\_\_\_ Full-time \_\_\_\_\_ Hours per week

9. Check one of the following that best describes your computer use at work?

I do not use a computer at work. \_\_\_\_\_

I have used a computer at work for:

- A. Mostly for word processing and preparing spreadsheets. \_\_\_\_\_
- B. Mostly for writing computer programs. \_\_\_\_\_
- C. Any other - please list: \_\_\_\_\_  
\_\_\_\_\_

10. Check any of the following that may apply to you (you may check multiple answers).

The reason I took this course was:

- A. This course is required. \_\_\_\_\_yes \_\_\_\_\_no
- B. I am interested in learning more about computers. \_\_\_\_\_yes \_\_\_\_\_no
- C. I am interested in learning more about computer programming. \_\_\_\_\_yes \_\_\_\_\_no

11. In general, describe your ability to use a computer prior to enrolling in this course.
  
12. In general, describe your ability to write a computer program (in any programming language) prior to enrolling in this course.

APPENDIX G

BASIC STUDENT JOURNAL

STUDENT NAME \_\_\_\_\_ SS# \_\_\_\_\_ WEEK OF \_\_\_\_\_

Directions: The purpose of this journal is to gather data of your study and programming activities in a computer programming course. The information gathered in this student journal WILL NOT AFFECT YOUR GRADE in the course. Please respond as accurately and completely as possible.

Enter the date and time (to the nearest quarter hour). If you read your text, write program code or execute (run) your program by yourself, then check SELF study. If you read the text, write program code or execute (run) your program with other students or receive any outside help from a tutor, then check TEAM study. If one of the STUDY/PROGRAMMING ACTIVITIES listed below describes either your study or programming activities, then check the appropriate column. If none of the STUDY/PROGRAMMING ACTIVITIES listed below adequately describes either your study or programming activities, then please describe your activities in the column marked "OTHER COMMENTS". These journals will be completed in class every Thursday.

PLEASE CHECK ONLY ONE COLUMN FOR EACH LINE. USE MULTIPLE LINES IF NECESSARY.

STUDY --- PROGRAMMING ACTIVITIES

DATE	TIME HH:MM	SELF	TEAM	READING TEXT OR HANDOUTS	PREPARE AND WRITE PROG.	EXECUTE PROG. & CORRECT ERRORS	OTHER COMMENTS

If any activity involved TEAM STUDY with another student or if you received help from a tutor, then please list the person(s) name below. In the column labeled "RELATIONSHIP", describe their relationship to you, i.e., another student, school tutor, friend, etc. In the column labeled "VALUE TO YOU", using a scale of 1 to 5, circle the value that represents this person's contribution in helping you to understand programming.

TEAM MEMBER'S NAME	RELATIONSHIP	VALUE TO YOU				
		NOT VALUABLE	SOMEWHAT VALUABLE	VERY VALUABLE	VERY VALUABLE	VERY VALUABLE
		1	2	3	4	5
		1	2	3	4	5
		1	2	3	4	5

## COBOL STUDENT JOURNAL

**STUDENT NAME** \_\_\_\_\_ **SS#** \_\_\_\_\_ **WEEK OF** \_\_\_\_\_

**Directions:** The purpose of this journal is to gather data of your study and programming activities in a computer programming course. The information gathered in this student journal **WILL NOT AFFECT YOUR GRADE** in the course. Please respond as accurately and completely as possible.

Enter the date and time (to the nearest quarter hour). If you read your text, write program code or debug a syntax error in your program by yourself, then check **SELF** study. If you read the text, write program code or debug a syntax error in your program with other students or receive any outside help from a tutor, then check **TEAM** study. If one of the **STUDY/PROGRAMMING ACTIVITIES** listed below describes either your study or programming activities, then check the appropriate column. If none of the **STUDY/PROGRAMMING ACTIVITIES** listed below adequately describes either your study or programming activities, then please describe your activities in the column marked "**OTHER COMMENTS**". These journals will be completed in class every Friday.

**PLEASE CHECK ONLY ONE COLUMN FOR EACH LINE. USE MULTIPLE LINES IF NECESSARY.**

### STUDY --- PROGRAMMING ACTIVITIES

DATE	TIME HH:MM	SELF	TEAM	READING TEXT	PREPARE AND WRITE PROG.	CHECK & SYNTAX	EXECUTE AND LOGIC.	OTHER COMMENTS

If any activity involved **TEAM STUDY** with another student or if you received help from a tutor, then please list the person(s) name below. In the column labeled "**RELATIONSHIP**", describe their relationship to you, i.e., another student, school tutor, friend, etc. In the column labeled "**VALUE TO YOU**", using a scale of 1 to 5, circle the value that represents this person's contribution in helping you to understand programming.

TEAM MEMBER'S NAME	RELATIONSHIP	VALUE TO YOU				
		NOT VALUABLE	SOMEWHAT VALUABLE	VERY VALUABLE		
		1	2	3	4	5
		1	2	3	4	5
		1	2	3	4	5



## APPENDIX H

### COBOL PROGRAM DEBUGGING TEST

#### DIRECTIONS:

This part of the exam requires you to locate and correct various programming logic errors in a COBOL program. At the beginning of this debugging test, your instructor will distribute a floppy disk containing programs and data files, program documentation and an answer sheet to you. The floppy disk will contain:

1. a COBOL source code for each program, and
2. the input data files to be used by the programs.

Each program will be accompanied by the following documentation:

1. a printed copy of the program,
2. a printed copy of the input data file,
3. a description of the program requirements,
4. a description of the program logic error,
5. the current, incorrect printed outputs of the program, and
6. the required, correct printed outputs of the program.

Each of the COBOL source programs will contain one or more logic errors. Each logic error will cause the program to produce incorrect outputs, or results. None of the programs will contain any syntax or execution errors. All programs will execute, but will produce incorrect results.

For each program, you are to locate each program logic error and write a description of the **cause** of the error on your answer sheet. Then you are required to use the computer to edit and execute the program until the program will produce the desired outputs, or results. Your test grade will be based upon your ability:

1. to locate and correctly **describe** the logic **error** on your answer sheet, and
2. to correct the logic error and **successfully execute** the program from your disk to produce the correct results.

You may use and write on any of the printed documentation provided with your test. You also may use the computer to help you locate and find the error. An answer sheet will be provided so that you may describe the cause of program logic error and to list the debugging tools that you used to find and/or correct the error.

At the beginning of the test you will be given the program documentation and a disk containing all of the test programs and data files. When you complete the debugging requirements for each program, hold up your hand and a test administrator will collect your answer sheet. At that point, you may continue working on the next program. You may only work on one program at a time. Once you hand in an answer sheet, you may NOT edit or change a previous test program on your disk! Any program that is edited or changed after the answer sheet has been handed into the test administrator will receive zero points. However, if you are stuck on a particular program, you are encouraged to proceed to the next test program.

Each COBOL source program will contain the data name NOTHING and will DISPLAY SPACE UPON CRT in the PROCEDURE DIVISION. All printed outputs for the test programs will be directed to the file "A:PROGX.RPT" (where X represents your program number) and these printed outputs may be viewed by using SPFPC.

You will be limited to a maximum of one hour and twenty minutes to take the exam. At the end of the exam the instructor will collect your disk and any remaining answer sheets and program documentation.

**PROGRAM 1**                      Program Name: PROG1.CBL

**PROGRAM REQUIREMENTS**

This program will input a vendor data file, which contains the vendor's name, background information, current balance and Y.T.D. information. A traditional report should be prepared, which should include report headers, detail lines and total lines. Financial totals for all vendors should be accumulated for the current balance, Y.T.D Purchases and Y.T.D. Payments and a final total line should be printed at the end of the report.

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The report totals printed on the total line are incorrect. You are to find the location of the logic error that causes the incorrect report totals and correct the program so that the correct report totals are provided.

There is only one logic error in this program.

The complete program listing follows.

**Data file for Program 1: VENDOR.DAT****Record Layout**

Field Name	Data Type
Vendor Name	Alphanumeric
Vendor Address	Alphanumeric
Current Balance	Numeric
YTD Purchases	Numeric
YTD Payments	Numeric

Standish, INC.	0023P.O. BOX 13455	New York, NY 23157	000345600001000000019784
MacMillian Mfg.	0019745 8th Ave	Alberta, MM 63562	00006798004500000008933
J. Smith	09341 Barnes ST.	Pgh, PA 15234	000009600000008900234000
Dollars, INC.	945223 Fast Blvd.	Sands, CA 65357	000889020001445600008903

**Correct Program Output**

VENDOR REPORT					PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PUR.	YTD PMTS	BALANCE
23	Standish, INC.	P.O. BOX 13455	100.00	197.84	345.60
19	MacMillian Mfg.	745 8th Ave	4,500.00	89.33	67.98
934	J. Smith	1 Barnes ST.	.89	2,340.00	9.60
9452	Dollars, INC.	23 Fast Blvd.	144.56	89.03	889.02
*** VENDOR REPORT TOTALS ***			4,745.45	2,716.20	1,312.20

**Incorrect Program Output**

VENDOR REPORT					PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PUR.	YTD PMTS	BALANCE
23	Standish, INC.	P.O. BOX 13455	100.00	197.84	345.60
19	MacMillian Mfg.	745 8th Ave	4,500.00	89.33	67.98
934	J. Smith	1 Barnes ST.	.89	2,340.00	9.60
9452	Dollars, INC.	23 Fast Blvd.	144.56	89.03	889.02
*** VENDOR REPORT TOTALS ***			144.56	89.03	889.02

## SOURCE PROGRAM 1

IDENTIFICATION DIVISION.  
PROGRAM-ID. TESTQ1.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
FILE-CONTROL.

    SELECT INPUT-FILE  
        ORGANIZATION IS LINE SEQUENTIAL  
        ASSIGN TO "A:VENDOR.DAT".  
    SELECT PRINT-FILE  
        ORGANIZATION IS LINE SEQUENTIAL  
        ASSIGN TO "A:PROG1.RPT".

DATA DIVISION.

FILE SECTION.

FD INPUT-FILE

    LABEL RECORDS ARE STANDARD  
    RECORD CONTAINS 80 CHARACTERS  
    DATA RECORD IS INPUT-RECORD.

01 INPUT-RECORD                  PIC X(80).

FD PRINT-FILE

    LABEL RECORDS ARE STANDARD  
    RECORD CONTAINS 132 CHARACTERS  
    DATA RECORD IS PRINT-RECORD.

01 PRINT-RECORD                  PIC X(80).

WORKING-STORAGE SECTION.

01 NAME-ADDRESS-FILE-END.

    05 FLAG                      PIC X(4) VALUE "GO ".  
    05 NOTHING                  PIC X.  
    05 LINE-SPACES              PIC X(80) VALUE SPACES.

01 VENDOR-INFORMATION.

    05 VENDOR-NAME              PIC X(15).  
    05 VENDOR-NUMBER             PIC 9999.  
    05 VENDOR-ADDRESS           PIC X(15).  
    05 VENDOR-CITY-STATE-ZIP    PIC X(20).  
    05 VENDOR-BALANCE           PIC \$9(6)V99.  
    05 VENDOR-YTD-PURCHASES    PIC \$9(6)V99.  
    05 VENDOR-YTD-PAYMENTS     PIC \$9(6)V99.

01 DETAIL-LINE.

    05 VENDOR-NUMBER-DL        PIC ZZZZ.  
    05 FILLER                  PIC X(1).  
    05 VENDOR-NAME-DL          PIC X(15).  
    05 FILLER                  PIC X(1).  
    05 VENDOR-ADDRESS-DL       PIC X(15).  
    05 FILLER                  PIC X(1).  
    05 VENDOR-YTD-PURCHASES-DL PIC ZZ,ZZZ.99.  
    05 FILLER                  PIC X(1).  
    05 VENDOR-YTD-PAYMENTS-DL  PIC ZZ,ZZZ.99.  
    05 FILLER                  PIC X(1).  
    05 VENDOR-BALANCE-DL       PIC ---,---.99.

01 HEADER-LINE-1.

    05 FILLER                  PIC X(26).  
    05 FILLER                  PIC X(20) VALUE "VENDOR REPORT".  
    05 FILLER                  PIC X(12).  
    05 FILLER                  PIC X(6) VALUE "PAGE: ".  
    05 PAGE-NUMBER-OUT          PIC ZZ.

01 HEADER-LINE-2.

    05 FILLER                  PIC X(5) VALUE "VEN.#".  
    05 FILLER                  PIC X(1).  
    05 FILLER                  PIC X(13) VALUE "VENDOR NAME".  
    05 FILLER                  PIC X(01).  
    05 FILLER                  PIC X(17) VALUE " ADDRESS".  
    05 FILLER                  PIC X(1).  
    05 FILLER                  PIC X(8) VALUE "YTD PUR.".   
    05 FILLER                  PIC X(1).  
    05 FILLER                  PIC X(9) VALUE " YTD PMTS".  
    05 FILLER                  PIC X(1).  
    05 FILLER                  PIC X(11) VALUE " BALANCE".

```

01 TOTAL-LINE.
   05 FILLER                PIC X(05).
   05 FILLER                PIC X(30)
                               VALUE **** VENDOR REPORT TOTALS ****.
   05 FILLER                PIC X(2).
   05 VENDOR-YTD-PURCHASES-TL PIC Z2,ZZZ.99.
   05 FILLER                PIC X(1).
   05 VENDOR-YTD-PAYMENTS-TL PIC Z2,ZZZ.99.
   05 FILLER                PIC X(1).
   05 VENDOR-BALANCE-TL     PIC ---,---.99.
01 ACCUMULATORS.
   05 TOTAL-YTD-PURCHASES    PIC S9(6)V99.
   05 TOTAL-YTD-PAYMENTS    PIC S9(6)V99.
   05 TOTAL-BALANCE         PIC S9(9)V99.
01 PAGE-CONTROL.
   05 PAGE-COUNT            PIC 999.
   05 LINE-COUNT           PIC 999.
   05 PAGE-SIZE            PIC 999.

```

## PROCEDURE DIVISION.

## START-HERE.

```

DISPLAY SPACE UPON CRT.
PERFORM INITIALIZE-VALUES.
PERFORM OPEN-FILES.
PERFORM READ-RECORD.
PERFORM PROCESS-REPORT UNTIL FLAG = "STOP".
PERFORM ACCUMULATE-TOTALS.
PERFORM PRINT-TOTALS.
PERFORM CLOSE-FILES.
STOP RUN.

```

## INITIALIZE-VALUES.

```

MOVE "GO " TO FLAG.
MOVE ZEROES TO TOTAL-YTD-PURCHASES.
MOVE ZEROES TO TOTAL-YTD-PAYMENTS.
MOVE ZEROES TO TOTAL-BALANCE.
MOVE 1 TO PAGE-COUNT.
MOVE 999 TO LINE-COUNT.
MOVE 20 TO PAGE-SIZE.

```

## PROCESS-HEADERS.

```

MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.
WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.
WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
WRITE PRINT-RECORD FROM HEADER-LINE-2 AFTER ADVANCING 1.
ADD 1 TO PAGE-COUNT.
MOVE ZEROES TO LINE-COUNT.

```

## PRINT-TOTALS.

```

MOVE TOTAL-BALANCE          TO VENDOR-BALANCE-TL.
MOVE TOTAL-YTD-PURCHASES    TO VENDOR-YTD-PURCHASES-TL.
MOVE TOTAL-YTD-PAYMENTS     TO VENDOR-YTD-PAYMENTS-TL.
WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
WRITE PRINT-RECORD FROM TOTAL-LINE AFTER ADVANCING 1.

```

## PROCESS-REPORT.

```

IF LINE-COUNT > PAGE-SIZE
  PERFORM PROCESS-HEADERS.
PERFORM MOVE-DATA.
PERFORM WRITE-RECORD.
PERFORM TOTAL-LINES.
PERFORM READ-RECORD.

```

## WRITE-RECORD.

```

WRITE PRINT-RECORD FROM DETAIL-LINE AFTER ADVANCING 1.

```

## ACCUMULATE-TOTALS.

```

ADD VENDOR-BALANCE          TO TOTAL-BALANCE.
ADD VENDOR-YTD-PURCHASES    TO TOTAL-YTD-PURCHASES.
ADD VENDOR-YTD-PAYMENTS     TO TOTAL-YTD-PAYMENTS.

```

MOVE-DATA.  
MOVE VENDOR-NUMBER TO VENDOR-NUMBER-DL.  
MOVE VENDOR-NAME TO VENDOR-NAME-DL.  
MOVE VENDOR-ADDRESS TO VENDOR-ADDRESS-DL.  
MOVE VENDOR-BALANCE TO VENDOR-BALANCE-DL.  
MOVE VENDOR-YTD-PURCHASES TO VENDOR-YTD-PURCHASES-DL.  
MOVE VENDOR-YTD-PAYMENTS TO VENDOR-YTD-PAYMENTS-DL.

TOTAL-LINES.  
ADD 1 TO LINE-COUNT.

READ-RECORD.  
READ INPUT-FILE INTO VENDOR-INFORMATION  
AT END MOVE "STOP" TO FLAG.

OPEN-FILES.  
OPEN INPUT INPUT-FILE.  
OPEN OUTPUT PRINT-FILE.

CLOSE-FILES.  
CLOSE INPUT-FILE  
PRINT-FILE.

**PROGRAM 2**Program Name: PROG2.CBL**PROGRAM DESCRIPTION**

This program will input a student data file, which contains the student's name, credits earned to date and the total quality points earned to date. A traditional report should be prepared, which should include report headers, detail lines and total lines reporting the average student Q.P.A. Each detail line should report the individual student's Q.P.A. An individual student's

Q.P.A. is calculated by the following formula:

$$\text{STUDENT Q.P.A.} = \frac{\text{STUDENT QUALITY POINTS}}{\text{STUDENT CREDITS TO DATE}}$$

Total student quality points and total student credits to date should be accumulated for all students and the average student Q.P.A. should be calculated by using the following formula and reported on the final total line.

$$\text{AVERAGE STUDENT Q.P.A.} = \frac{\text{TOTAL STUDENT QUALITY POINTS}}{\text{TOTAL STUDENT CREDITS TO DATE}}$$

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The individual student Q.P.A. is incorrect, but the average student Q.P.A. is correct. You are to find the location of the logic error that causes the incorrect student Q.P.A. and correct the program so that the correct report totals are provided.

There is only one logic error in this program.

The complete program listing follows.



Input Data file for Program 2: STUDENT.DAT

## Record Layout

Field Name	Data Type
Student Name	Alphanumeric
Student Number	Numeric
Quality Points	Numeric
Total Credits	Numeric

Mary Jones	001400400010
Doug Smith	002100300015
Adam Upp	000200200012
Bart Simpson	000700300010

Correct Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS	QUAL.PTS	QPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
2	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00
*** AVERAGE STUDENT QPA ***				2.55

Incorrect Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS	QUAL.PTS	QPA
14	Mary Jones	10	40	
21	Doug Smith	15	30	4.00
2	Adam Upp	12	20	2.00
7	Bart Simpson	10	30	1.66
*** AVERAGE STUDENT QPA ***				2.55

## SOURCE PROGRAM TWO

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TESTQ2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE
    ORGANIZATION IS LINE SEQUENTIAL
    ASSIGN TO "C:STUDENTD.DAT".
    SELECT PRINT-FILE
    ORGANIZATION IS LINE SEQUENTIAL
    ASSIGN TO "C:PROG2.RPT".

DATA DIVISION.
FILE SECTION.
FD INPUT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS INPUT-RECORD.
01 INPUT-RECORD          PIC X(80).
FD PRINT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS
    DATA RECORD IS PRINT-RECORD.
01 PRINT-RECORD          PIC X(80).

WORKING-STORAGE SECTION.
01 NAME-ADDRESS-FILE-END.
    05 FLAG                PIC X(4) VALUE "GO ".
    05 NOTHING             PIC X.
    05 LINE-SPACES         PIC X(80) VALUE SPACES.
01 STUDENT-INFORMATION.
    05 STUDENT-NAME        PIC X(20).
    05 STUDENT-NUMBER      PIC 9999.
    05 STUDENT-QUALITY-POINTS PIC 9(4).
    05 STUDENT-CREDITS-TO-DATE PIC 9(4).
01 DETAIL-LINE.
    05 STUDENT-NUMBER-DL   PIC ZZZZ.
    05 FILLER              PIC X(6).
    05 STUDENT-NAME-DL    PIC X(20).
    05 FILLER              PIC X(2).
    05 STUDENT-CREDITS-TO-DATE-DL PIC ZZZZ.
    05 FILLER              PIC X(11).
    05 STUDENT-QUALITY-POINTS-DL PIC ZZZZ.
    05 FILLER              PIC X(10).
    05 STUDENT-QPA-DL     PIC Z.99.
01 HEADER-LINE-1.
    05 FILLER              PIC X(17).
    05 FILLER              PIC X(20) VALUE "STUDENT QPA REPORT".
    05 FILLER              PIC X(19).
    05 FILLER              PIC X(6) VALUE "PAGE: ".
    05 PAGE-NUMBER-OUT    PIC ZZ.
01 HEADER-LINE-2.
    05 FILLER              PIC X(8) VALUE "STUD. #".
    05 FILLER              PIC X(3).
    05 FILLER              PIC X(13) VALUE "STUDENT NAME".
    05 FILLER              PIC X(08).
    05 FILLER              PIC X(07) VALUE "CREDITS".
    05 FILLER              PIC X(07).
    05 FILLER              PIC X(08) VALUE "QUAL.PTS".
    05 FILLER              PIC X(08).
    05 FILLER              PIC X(08) VALUE "QPA".
01 TOTAL-LINE.
    05 FILLER              PIC X(18).
    05 FILLER              PIC X(30)
    VALUE "**** AVERAGE STUDENT QPA ****".
    05 FILLER              PIC X(13).
    05 STUDENT-AVERAGE-QPA-TL PIC Z.99.

```

```

01 ACCUMULATORS.
   05 TOTAL-QUALITY-POINTS PIC S9(6)V99.
   05 TOTAL-CREDITS-TO-DATE PIC S9(6)V99.
01 PAGE-CONTROL.
   05 PAGE-COUNT PIC 999.
   05 LINE-COUNT PIC 999.
   05 PAGE-SIZE PIC 999.

PROCEDURE DIVISION.
START-HERE.
  DISPLAY SPACE UPON CRT.
  PERFORM INITIALIZE-VALUES.
  PERFORM OPEN-FILES.
  PERFORM READ-RECORD.
  PERFORM PROCESS-REPORT UNTIL FLAG = "STOP".
  PERFORM PRINT-TOTALS.
  PERFORM CLOSE-FILES.
  STOP RUN.

INITIALIZE-VALUES.
  MOVE "GO " TO FLAG.
  MOVE ZEROES TO TOTAL-QUALITY-POINTS.
  MOVE ZEROES TO TOTAL-CREDITS-TO-DATE.
  MOVE 1 TO PAGE-COUNT.
  MOVE 999 TO LINE-COUNT.
  MOVE 20 TO PAGE-SIZE.

PROCESS-HEADERS.
  MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.
  WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.
  WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
  WRITE PRINT-RECORD FROM HEADER-LINE-2 AFTER ADVANCING 1.
  ADD 1 TO PAGE-COUNT.
  MOVE ZEROES TO LINE-COUNT.

PRINT-TOTALS.
  COMPUTE STUDENT-AVERAGE-QPA-TL =
    TOTAL-QUALITY-POINTS/TOTAL-CREDITS-TO-DATE.
  WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
  WRITE PRINT-RECORD FROM TOTAL-LINE AFTER ADVANCING 1.

PROCESS-REPORT.
  IF LINE-COUNT > PAGE-SIZE
    PERFORM PROCESS-HEADERS.
  PERFORM MOVE-DATA.
  PERFORM ACCUMULATE-TOTALS.
  PERFORM WRITE-RECORD.
  PERFORM CALCULATE-QPA.
  PERFORM TOTAL-LINES-PRINTED.
  PERFORM READ-RECORD.

WRITE-RECORD.
  WRITE PRINT-RECORD FROM DETAIL-LINE AFTER ADVANCING 1.

ACCUMULATE-TOTALS.
  ADD STUDENT-QUALITY-POINTS TO TOTAL-QUALITY-POINTS.
  ADD STUDENT-CREDITS-TO-DATE TO TOTAL-CREDITS-TO-DATE.

MOVE-DATA.
  MOVE STUDENT-NUMBER TO STUDENT-NUMBER-DL.
  MOVE STUDENT-NAME TO STUDENT-NAME-DL.
  MOVE STUDENT-QUALITY-POINTS TO STUDENT-QUALITY-POINTS-DL.
  MOVE STUDENT-CREDITS-TO-DATE TO
    STUDENT-CREDITS-TO-DATE-DL.

CALCULATE-QPA.
  COMPUTE STUDENT-QPA-DL =
    STUDENT-QUALITY-POINTS/ STUDENT-CREDITS-TO-DATE.

TOTAL-LINES-PRINTED.
  ADD 1 TO LINE-COUNT.

```

READ-RECORD.  
 READ INPUT-FILE INTO STUDENT-INFORMATION  
 AT END MOVE "STOP" TO FLAG.

OPEN-FILES.  
 OPEN INPUT INPUT-FILE.  
 OPEN OUTPUT PRINT-FILE.

CLOSE-FILES.  
 CLOSE INPUT-FILE  
 PRINT-FILE.

**PROGRAM 3****Program Name: PROG3.CBL****PROGRAM DESCRIPTION**

This program will input a student data file, which contains the student's name, credits earned to date and the total quality points earned to date. A high/low analysis report should be prepared, which should include report headers, detail lines and final report lines reporting the name of the student and their Q.P.A., who received either the highest or the lowest Q.P.A. Each detail line should report the individual student's Q.P.A. An individual student's Q.P.A. is calculated by the following formula:

$$\text{STUDENT Q.P.A.} = \frac{\text{STUDENT QUALITY POINTS}}{\text{STUDENT CREDITS TO DATE}}$$

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The analysis lines provided at the end of the student report, which lists the student with the highest and lowest Q.P.A., are incorrect. You are to find the location of the logic error that causes the program to report the incorrect highest Q.P.A. and lowest Q.P.A.. You are to correct the program so that the correct highest and lowest Q.P.A. with corresponding student's name are provided.

There is only one logic error in this program.

The complete program listing follows.

Input Data file for Program 3: STUDENTD.DAT

## Record Layout

Field Name	Data Type
Student Name	Alphanumeric
Student Number	Numeric
Quality Points	Numeric
Total Credits	Numeric

Mary Jones	001400400010
Doug Smith	002100300015
Adam Upp	000200200012
Bart Simpson	000700300010

Correct Program Output

STUDENT GPA REPORT					PAGE: 1
STUD. #	STUDENT NAME	CREDITS	QUAL.PTS	GPA	
14	Mary Jones	10	40	4.00	
21	Doug Smith	15	30	2.00	
2	Adam Upp	12	20	1.66	
7	Bart Simpson	10	30	3.00	
Mary Jones	HAS THE HIGHEST AVERAGE OF			4.00	
Adam Upp	HAS THE LOWEST AVERAGE OF			1.66	

Incorrect Program Output

STUDENT GPA REPORT					PAGE: 1
STUD. #	STUDENT NAME	CREDITS	QUAL.PTS	GPA	
14	Mary Jones	10	40	4.00	
21	Doug Smith	15	30	2.00	
2	Adam Upp	12	20	1.66	
7	Bart Simpson	10	30	3.00	
Bart Simpson	HAS THE HIGHEST AVERAGE OF			3.00	
Bart Simpson	HAS THE LOWEST AVERAGE OF			3.00	

## SOURCE PROGRAM 3

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TESTQ3.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE
        ORGANIZATION IS LINE SEQUENTIAL
        ASSIGN TO "A:STUDENTD.DAT".
    SELECT PRINT-FILE
        ORGANIZATION IS LINE SEQUENTIAL
        ASSIGN TO "A:PROG3.RPT".

DATA DIVISION.
FILE SECTION.
FD INPUT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS INPUT-RECORD.
01 INPUT-RECORD          PIC X(80).
FD PRINT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS
    DATA RECORD IS PRINT-RECORD.
01 PRINT-RECORD          PIC X(80).

WORKING-STORAGE SECTION.
01 NAME-ADDRESS-FILE-END.
    05 FLAG                PIC X(4) VALUE "GO ".
    05 NOTHING              PIC X.
    05 LINE-SPACES          PIC X(80) VALUE SPACES.
    05 QPA-MATH              PIC 9V99 VALUE ZERO.

01 STUDENT-INFORMATION.
    05 STUDENT-NAME          PIC X(20).
    05 STUDENT-NUMBER        PIC 9999.
    05 STUDENT-QUALITY-POINTS PIC 9(4).
    05 STUDENT-CREDITS-TO-DATE PIC 9(4).

01 DETAIL-LINE.
    05 STUDENT-NUMBER-DL     PIC ZZZZ.
    05 FILLER                PIC X(6).
    05 STUDENT-NAME-DL       PIC X(20).
    05 FILLER                PIC X(2).
    05 STUDENT-CREDITS-TO-DATE-DL PIC ZZZZ.
    05 FILLER                PIC X(11).
    05 STUDENT-QUALITY-POINTS-DL PIC ZZZZ.
    05 FILLER                PIC X(10).
    05 STUDENT-QPA-DL        PIC Z.99.

01 HEADER-LINE-1.
    05 FILLER                PIC X(17).
    05 FILLER                PIC X(20) VALUE "STUDENT QPA REPORT".
    05 FILLER                PIC X(19).
    05 FILLER                PIC X(6) VALUE "PAGE: ".
    05 PAGE-NUMBER-OUT       PIC ZZ.

01 HEADER-LINE-2.
    05 FILLER                PIC X(8) VALUE "STUD. #".
    05 FILLER                PIC X(3).
    05 FILLER                PIC X(13) VALUE "STUDENT NAME".
    05 FILLER                PIC X(08).
    05 FILLER                PIC X(07) VALUE "CREDITS".
    05 FILLER                PIC X(07).
    05 FILLER                PIC X(08) VALUE "QUAL.PTS".
    05 FILLER                PIC X(08).
    05 FILLER                PIC X(08) VALUE "QPA".

```

```

01 HIGH-LOW-FIELDS.
   05 HIGH-QPA          PIC 999V99.
   05 LOW-QPA           PIC 999V99.
   05 HIGH-STUDENT-NAME PIC X(20).
   05 LOW-STUDENT-NAME  PIC X(20).
01 HIGH-LINE.
   05 HIGH-STUDENT-NAME-HL PIC X(20).
   05 FILER                PIC X(28) VALUE
      " HAS THE HIGHEST AVERAGE OF ".
   05 HIGH-QPA-HL         PIC ZZZ.99.
01 LOW-LINE.
   05 LOW-STUDENT-NAME-LL PIC X(20).
   05 FILER                PIC X(28) VALUE
      " HAS THE LOWEST AVERAGE OF ".
   05 LOW-QPA-LL         PIC ZZZ.99.
01 PAGE-CONTROL.
   05 PAGE-COUNT         PIC 999.
   05 LINE-COUNT        PIC 999.
   05 PAGE-SIZE         PIC 999.

```

## PROCEDURE DIVISION.

## START-HERE.

```

DISPLAY SPACE UPON CRT.
PERFORM INITIALIZE-VALUES.
PERFORM OPEN-FILES.
PERFORM READ-RECORD.
PERFORM PROCESS-REPORT UNTIL FLAG = "STOP".
PERFORM FIND-HIGH-QPA.
PERFORM FIND-LOW-QPA.
PERFORM PRINT-HIGH-LOW.
PERFORM CLOSE-FILES.
STOP RUN.

```

## INITIALIZE-VALUES.

```

MOVE "GO " TO FLAG.
MOVE 1 TO PAGE-COUNT.
MOVE 999 TO LINE-COUNT.
MOVE 20 TO PAGE-SIZE.
MOVE ZEROES TO HIGH-QPA.
MOVE 999 TO LOW-QPA.

```

## PRINT-HIGH-LOW.

```

MOVE HIGH-QPA TO HIGH-QPA-HL.
MOVE HIGH-STUDENT-NAME TO HIGH-STUDENT-NAME-HL.
MOVE LOW-QPA TO LOW-QPA-LL.
MOVE LOW-STUDENT-NAME TO LOW-STUDENT-NAME-LL.
WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
WRITE PRINT-RECORD FROM HIGH-LINE AFTER ADVANCING 1 LINES.
WRITE PRINT-RECORD FROM LOW-LINE AFTER ADVANCING 1 LINES.

```

## PROCESS-HEADERS.

```

MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.
WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.
WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
WRITE PRINT-RECORD FROM HEADER-LINE-2 AFTER ADVANCING 1.
ADD 1 TO PAGE-COUNT.
MOVE ZEROES TO LINE-COUNT.

```

## PROCESS-REPORT.

```

IF LINE-COUNT > PAGE-SIZE
  PERFORM PROCESS-HEADERS.
PERFORM MOVE-DATA.
PERFORM CALCULATE-QPA.
PERFORM WRITE-RECORD.
PERFORM TOTAL-LINES-PRINTED.
PERFORM READ-RECORD.

```

## WRITE-RECORD.

```

WRITE PRINT-RECORD FROM DETAIL-LINE AFTER ADVANCING 1.

```



```
FIND-HIGH-QPA.  
  IF HIGH-QPA < QPA-MATH  
    MOVE QPA-MATH TO HIGH-QPA  
    MOVE STUDENT-NAME TO HIGH-STUDENT-NAME.  
  
FIND-LOW-QPA.  
  IF LOW-QPA > QPA-MATH  
    MOVE QPA-MATH TO LOW-QPA  
    MOVE STUDENT-NAME TO LOW-STUDENT-NAME.  
  
MOVE-DATA.  
  MOVE STUDENT-NUMBER TO STUDENT-NUMBER-DL.  
  MOVE STUDENT-NAME TO STUDENT-NAME-DL.  
  MOVE STUDENT-QUALITY-POINTS TO STUDENT-QUALITY-POINTS-DL.  
  MOVE STUDENT-CREDITS-TO-DATE TO  
    STUDENT-CREDITS-TO-DATE-DL.  
  
CALCULATE-QPA.  
  COMPUTE STUDENT-QPA-DL QPA-MATH =  
    STUDENT-QUALITY-POINTS/ STUDENT-CREDITS-TO-DATE.  
  
TOTAL-LINES-PRINTED.  
  ADD 1 TO LINE-COUNT.  
  
READ-RECORD.  
  READ INPUT-FILE INTO STUDENT-INFORMATION  
  AT END MOVE "STOP" TO FLAG.  
  
OPEN-FILES.  
  OPEN INPUT INPUT-FILE.  
  OPEN OUTPUT PRINT-FILE.  
  
CLOSE-FILES.  
  CLOSE INPUT-FILE  
  PRINT-FILE.
```

**PROGRAM 4**                    Program Name: PROG4.CBL

**PROGRAM DESCRIPTION**

This program will input a student data file, which contains the student's name, credits earned to date and the total quality points earned to date. A high/low analysis report should be prepared, which should include report headers, detail lines and final report lines reporting the name of the student and their Q.P.A., who received either the highest or the lowest Q.P.A. Each detail line should report the individual student's Q.P.A. An individual student's Q.P.A. is calculated by the following formula:

$$\text{STUDENT Q.P.A.} = \frac{\text{STUDENT QUALITY POINTS}}{\text{STUDENT CREDITS TO DATE}}$$

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The analysis lines provided at the end of the student report, which lists the student with the highest and lowest Q.P.A., are incorrect. You are to find the location of the logic error that causes the program to report the incorrect highest Q.P.A. and lowest Q.P.A.. You are to correct the program so that the correct highest and lowest Q.P.A. with corresponding student's name are provided.

There are TWO logic errors in this program.

The complete program listing follows.

Input Data file for Program 4: STUDENT.DAT

## Record Layout

Field Name	Data Type
Student Name	Alphanumeric
Student Number	Numeric
Quality Points	Numeric
Total Credits	Numeric

Mary Jones	001400400010
Doug Smith	002100300015
Adam Upp	000200200012
Bart Simpson	000700300010

Correct Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS	QUAL.PTS	QPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
2	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00

Mary Jones HAS THE HIGHEST AVERAGE OF 4.00  
Adam Upp HAS THE LOWEST AVERAGE OF 1.66

Incorrect Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS	QUAL.PTS	QPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
2	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00

Mary Jones HAS THE HIGHEST AVERAGE OF .00  
Mary Jones HAS THE LOWEST AVERAGE OF .00

## SOURCE PROGRAM 4

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TESTQ4.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE
        ORGANIZATION IS LINE SEQUENTIAL
        ASSIGN TO "A:STUDENTD.DAT".
    SELECT PRINT-FILE
        ORGANIZATION IS LINE SEQUENTIAL
        ASSIGN TO "A:PROG4.RPT".

DATA DIVISION.
FILE SECTION.
FD INPUT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS INPUT-RECORD.
01 INPUT-RECORD          PIC X(80).
FD PRINT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS
    DATA RECORD IS PRINT-RECORD.
01 PRINT-RECORD         PIC X(80).

WORKING-STORAGE SECTION.
01 NAME-ADDRESS-FILE-END.
    05 FLAG              PIC X(4) VALUE "GO ".
    05 NOTHING           PIC X.
    05 LINE-SPACES      PIC X(80) VALUE SPACES.
    05 QPA-MATH         PIC 9V99 VALUE ZERO.

01 STUDENT-INFORMATION.
    05 STUDENT-NAME      PIC X(20).
    05 STUDENT-NUMBER    PIC 9999.
    05 STUDENT-QUALITY-POINTS PIC 9(4).
    05 STUDENT-CREDITS-TO-DATE PIC 9(4).

01 DETAIL-LINE.
    05 STUDENT-NUMBER-DL PIC ZZZZ.
    05 FILLER            PIC X(6).
    05 STUDENT-NAME-DL   PIC X(20).
    05 FILLER            PIC X(2).
    05 STUDENT-CREDITS-TO-DATE-DL PIC ZZZZ.
    05 FILLER            PIC X(11).
    05 STUDENT-QUALITY-POINTS-DL PIC ZZZZ.
    05 FILLER            PIC X(10).
    05 STUDENT-QPA-DL    PIC Z.99.

01 HEADER-LINE-1.
    05 FILLER            PIC X(17).
    05 FILLER            PIC X(20) VALUE "STUDENT QPA REPORT".
    05 FILLER            PIC X(19).
    05 FILLER            PIC X(6) VALUE "PAGE: ".
    05 PAGE-NUMBER-OUT   PIC ZZ.

01 HEADER-LINE-2.
    05 FILLER            PIC X(8) VALUE "STUD. #".
    05 FILLER            PIC X(3).
    05 FILLER            PIC X(13) VALUE "STUDENT NAME".
    05 FILLER            PIC X(08).
    05 FILLER            PIC X(07) VALUE "CREDITS".
    05 FILLER            PIC X(07).
    05 FILLER            PIC X(08) VALUE "QUAL.PTS".
    05 FILLER            PIC X(08).
    05 FILLER            PIC X(08) VALUE "QPA".

```

```

01 HIGH-LOW-FIELDS.
   05 HIGH-QPA          PIC 999V99.
   05 LOW-QPA           PIC 999V99.
   05 HIGH-STUDENT-NAME PIC X(20).
   05 LOW-STUDENT-NAME  PIC X(20).

01 HIGH-LINE.
   05 HIGH-STUDENT-NAME-HL PIC X(20).
   05 FILER                 PIC X(28) VALUE
      " HAS THE HIGHEST AVERAGE OF ".
   05 HIGH-QPA-HL          PIC ZZZ.99.

01 LOW-LINE.
   05 LOW-STUDENT-NAME-LL  PIC X(20).
   05 FILER                 PIC X(28) VALUE
      " HAS THE LOWEST AVERAGE OF ".
   05 LOW-QPA-LL          PIC ZZZ.99.

01 PAGE-CONTROL.
   05 PAGE-COUNT          PIC 999.
   05 LINE-COUNT          PIC 999.
   05 PAGE-SIZE           PIC 999.

```

PROCEDURE DIVISION.

```

START-HERE.
  DISPLAY SPACE UPON CRT.
  PERFORM INITIALIZE-VALUES.
  PERFORM OPEN-FILES.
  PERFORM READ-RECORD.
  PERFORM PROCESS-REPORT UNTIL FLAG = "STOP".
  PERFORM PRINT-HIGH-LOW.
  PERFORM CLOSE-FILES.
  STOP RUN.

```

```

PROCESS-REPORT.
  IF LINE-COUNT > PAGE-SIZE
    PERFORM PROCESS-HEADERS.
  PERFORM MOVE-DATA.
  PERFORM FIND-HIGH-QPA.
  PERFORM FIND-LOW-QPA.
  PERFORM CALCULATE-QPA.
  PERFORM WRITE-RECORD.
  PERFORM TOTAL-LINES-PRINTED.
  PERFORM READ-RECORD.

```

```

INITIALIZE-VALUES.
  MOVE "GO " TO FLAG.
  MOVE 1 TO PAGE-COUNT.
  MOVE 999 TO LINE-COUNT.
  MOVE 20 TO PAGE-SIZE.
  MOVE ZEROES TO HIGH-QPA.
  MOVE 999 TO LOW-QPA.

```

```

PRINT-HIGH-LOW.
  MOVE HIGH-QPA          TO HIGH-QPA-HL.
  MOVE HIGH-STUDENT-NAME TO HIGH-STUDENT-NAME-HL.
  MOVE LOW-QPA          TO LOW-QPA-LL.
  MOVE LOW-STUDENT-NAME TO LOW-STUDENT-NAME-LL.
  WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
  WRITE PRINT-RECORD FROM HIGH-LINE AFTER ADVANCING 1 LINES.
  WRITE PRINT-RECORD FROM LOW-LINE AFTER ADVANCING 1 LINES.

```

```

PROCESS-HEADERS.
  MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.
  WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.
  WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.
  WRITE PRINT-RECORD FROM HEADER-LINE-2 AFTER ADVANCING 1.
  ADD 1 TO PAGE-COUNT.
  MOVE ZEROES TO LINE-COUNT.

```

```

WRITE-RECORD.
  WRITE PRINT-RECORD FROM DETAIL-LINE AFTER ADVANCING 1.

```

```
MOVE-DATA.  
  MOVE STUDENT-NUMBER      TO STUDENT-NUMBER-DL.  
  MOVE STUDENT-NAME       TO STUDENT-NAME-DL.  
  MOVE STUDENT-QUALITY-POINTS TO STUDENT-QUALITY-POINTS-DL.  
  MOVE STUDENT-CREDITS-TO-DATE TO STUDENT-CREDITS-TO-DATE-DL.  
  
CALCULATE-QPA.  
  COMPUTE STUDENT-QPA-DL QPA-MATH =  
    STUDENT-QUALITY-POINTS/ STUDENT-CREDITS-TO-DATE.  
  
TOTAL-LINES-PRINTED.  
  ADD 1 TO LINE-COUNT.  
  
FIND-HIGH-QPA.  
  IF HIGH-QPA > QPA-MATH  
    MOVE QPA-MATH TO HIGH-QPA  
    MOVE STUDENT-NAME TO HIGH-STUDENT-NAME.  
  
FIND-LOW-QPA.  
  IF LOW-QPA > QPA-MATH  
    MOVE QPA-MATH TO LOW-QPA  
    MOVE STUDENT-NAME TO LOW-STUDENT-NAME.  
  
READ-RECORD.  
  READ INPUT-FILE INTO STUDENT-INFORMATION  
  AT END MOVE "STOP" TO FLAG.  
  
OPEN-FILES.  
  OPEN INPUT INPUT-FILE.  
  OPEN OUTPUT PRINT-FILE.  
  
CLOSE-FILES.  
  CLOSE INPUT-FILE  
  PRINT-FILE.
```

**PROGRAM 5****Program Name: PROG5.CBL****PROGRAM REQUIREMENTS**

This program will input a vendor data file, which contains the vendor's name, background information, current balance and Y.T.D. information. A traditional report should be prepared, which should include report headers, detail lines and total lines. Financial totals for all vendors should be accumulated for the current balance, Y.T.D Purchases and Y.T.D. Payments and a final total line should be printed at the end of the report. Following the final total line a distribution analysis report should appear, listing the dollar amount and percentage of the vendor's outstanding balance "under 500 dollars" and "over 500 dollars" due.

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The total balance printed on the total line are incorrect. In addition, the results of the distribution analysis are incorrect. You are to find the location of the logic errors that causes the incorrect total balance and distribution report and correct the program so that the correct distribution report and total balances are provided.

There is only one logic error in this program.

The complete program listing follows.

Data file for Program 5: VENDOR.DAT

## Record Layout

Field Name	Data Type
Vendor Name	Alphanumeric
Vendor Address	Alphanumeric
Current Balance	Numeric
YTD Purchases	Numeric
YTD Payments	Numeric

Standish, INC. 0023P.O. BOX 13455	New York, NY 23157	000345600001000000019784
MacMillian Mfg. 0019745 8th Ave	Alberta, NM 63562	00006798004500000008933
J. Smith 09341 Barnes ST.	Pgh, PA 15234	000009600000008900234000
Dollars, INC. 945223 Fast Blvd.	Sands, CA 65357	000889020001445600008903

Correct Program Output

VENDOR REPORT					PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PUR.	YTD PMTS	BALANCE
23	Standish, INC.	P.O. BOX 13455	100.00	197.84	345.60
19	MacMillian Mfg.	745 8th Ave	4,500.00	89.33	67.98
934	J. Smith	1 Barnes ST.	.89	2,340.00	9.60
9452	Dollars, INC.	23 Fast Blvd.	144.56	89.03	889.02
*** VENDOR REPORT TOTALS ***			4,745.45	2,716.20	1,312.20

DISTRIBUTION ANALYSIS REPORT		
	DOLLARS	PERCENT
UNDER 500 DOLLARS BALANCE	423.18	32.24
OVER 500 DOLLARS BALANCE	889.02	67.75

Incorrect Program Output

VENDOR REPORT					PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PUR.	YTD PMTS	BALANCE
23	Standish, INC.	P.O. BOX 13455	100.00	197.84	345.60
19	MacMillian Mfg.	745 8th Ave	4,500.00	89.33	67.98
934	J. Smith	1 Barnes ST.	.89	2,340.00	9.60
9452	Dollars, INC.	23 Fast Blvd.	144.56	89.03	889.02
*** VENDOR REPORT TOTALS ***			4,745.45	2,716.20	1,735.38

DISTRIBUTION ANALYSIS REPORT		
	DOLLARS	PERCENT
UNDER 500 DOLLARS BALANCE	423.18	24.38
OVER 500 DOLLARS BALANCE	889.02	51.22



## SOURCE PROGRAM 5

IDENTIFICATION DIVISION.  
PROGRAM-ID. TESTQ5.  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
FILE-CONTROL.

SELECT INPUT-FILE  
ORGANIZATION IS LINE SEQUENTIAL  
ASSIGN TO "A.VENDOR.DAT".  
SELECT PRINT-FILE  
ORGANIZATION IS LINE SEQUENTIAL  
ASSIGN TO "A:PROG5.RPT".

DATA DIVISION.  
FILE SECTION.

FD INPUT-FILE

LABEL RECORDS ARE STANDARD  
RECORD CONTAINS 80 CHARACTERS  
DATA RECORD IS INPUT-RECORD.

01 INPUT-RECORD PIC X(80).

FD PRINT-FILE

LABEL RECORDS ARE STANDARD  
RECORD CONTAINS 132 CHARACTERS  
DATA RECORD IS PRINT-RECORD.

01 PRINT-RECORD PIC X(80).

WORKING-STORAGE SECTION.

01 NAME-ADDRESS-FILE-END.

05 FLAG PIC X(4) VALUE "GO ".  
05 NOTHING PIC X.  
05 LINE-SPACES PIC X(80) VALUE SPACES.

01 VENDOR-INFORMATION.

05 VENDOR-NAME PIC X(15).  
05 VENDOR-NUMBER PIC 9999.  
05 VENDOR-ADDRESS PIC X(15).  
05 VENDOR-CITY-STATE-ZIP PIC X(20).  
05 VENDOR-BALANCE PIC S9(6)V99.  
05 VENDOR-YTD-PURCHASES PIC S9(6)V99.  
05 VENDOR-YTD-PAYMENTS PIC S9(6)V99.

01 DETAIL-LINE.

05 VENDOR-NUMBER-DL PIC ZZZZ.  
05 FILLER PIC X(1).  
05 VENDOR-NAME-DL PIC X(15).  
05 FILLER PIC X(1).  
05 VENDOR-ADDRESS-DL PIC X(15).  
05 FILLER PIC X(1).  
05 VENDOR-YTD-PURCHASES-DL PIC Z2,ZZZ.99.  
05 FILLER PIC X(1).  
05 VENDOR-YTD-PAYMENTS-DL PIC Z2,ZZZ.99.  
05 FILLER PIC X(1).  
05 VENDOR-BALANCE-DL PIC ---,---.99.

01 HEADER-LINE-1.

05 FILLER PIC X(26).  
05 FILLER PIC X(20) VALUE "VENDOR REPORT".  
05 FILLER PIC X(12).  
05 FILLER PIC X(6) VALUE "PAGE: ".  
05 PAGE-NUMBER-OUT PIC ZZ.

01 HEADER-LINE-2.

05 FILLER PIC X(5) VALUE "VEN.#".  
05 FILLER PIC X(1).  
05 FILLER PIC X(13) VALUE "VENDOR NAME".  
05 FILLER PIC X(01).  
05 FILLER PIC X(17) VALUE " ADDRESS".  
05 FILLER PIC X(1).  
05 FILLER PIC X(8) VALUE "YTD PUR.".  
05 FILLER PIC X(1).  
05 FILLER PIC X(9) VALUE " YTD PMTS".  
05 FILLER PIC X(1).  
05 FILLER PIC X(11) VALUE " BALANCE".

```

01 TOTAL-LINE.
   05 FILLER                PIC X(05).
   05 FILLER                PIC X(30)
                                VALUE **** VENDOR REPORT TOTALS ****.
   05 FILLER                PIC X(2).
   05 VENDOR-YTD-PURCHASES-TL PIC Z2,ZZZ.99.
   05 FILLER                PIC X(1).
   05 VENDOR-YTD-PAYMENTS-TL PIC Z2,ZZZ.99.
   05 FILLER                PIC X(1).
   05 VENDOR-BALANCE-TL     PIC ---,---.99.

```

```

01 DISTRIBUTION-HEADER-1.
   05 FILLER                PIC X(30)
                                VALUE "DISTRIBUTION ANALYSIS REPORT ".

```

```

01 DISTRIBUTION-HEADER-2.
   05 FILLER                PIC X(32).
   05 FILLER                PIC X(07)
                                VALUE "DOLLARS".
   05 FILLER                PIC X(3).
   05 FILLER                PIC X(07)
                                VALUE "PERCENT".

```

```

01 DISTRIBUTION-OVER.
   05 FILLER                PIC X(30)
                                VALUE "OVER 500 DOLLARS BALANCE ".
   05 OVER-500-TL          PIC Z2,ZZZ.99.
   05 FILLER                PIC X(4).
   05 PRECENT-OVER-TL      PIC ZZZ.99.

```

```

01 DISTRIBUTION-UNDER.
   05 FILLER                PIC X(30)
                                VALUE "UNDER 500 DOLLARS BALANCE ".
   05 UNDER-500-TL        PIC Z2,ZZZ.99.
   05 FILLER                PIC X(4).
   05 PRECENT-UNDER-TL    PIC ZZZ.99.

```

```

01 ACCUMULATORS.
   05 TOTAL-YTD-PURCHASES  PIC S9(6)V99.
   05 TOTAL-YTD-PAYMENTS  PIC S9(6)V99.
   05 TOTAL-BALANCE        PIC S9(9)V99.
   05 OVER-500             PIC S9(9)V99.
   05 UNDER-500           PIC S9(9)V99.

```

```

01 PAGE-CONTROL.
   05 PAGE-COUNT           PIC 999.
   05 LINE-COUNT           PIC 999.
   05 PAGE-SIZE           PIC 999.

```

PROCEDURE DIVISION.

START-HERE.

```

DISPLAY SPACE UPON CRT.
PERFORM INITIALIZE-VALUES.
PERFORM OPEN-FILES.
PERFORM READ-RECORD.
PERFORM PROCESS-REPORT UNTIL FLAG = "STOP".
PERFORM PRINT-TOTALS.
PERFORM PRINT-PRECENT.
PERFORM CLOSE-FILES.
STOP RUN.

```

INITIALIZE-VALUES.

```

MOVE "GO " TO FLAG.
MOVE ZEROES TO TOTAL-YTD-PURCHASES.
MOVE ZEROES TO TOTAL-YTD-PAYMENTS.
MOVE ZEROES TO TOTAL-BALANCE.
MOVE ZEROES TO UNDER-500.
MOVE ZEROES TO OVER-500.
MOVE 1 TO PAGE-COUNT.
MOVE 999 TO LINE-COUNT.
MOVE 20 TO PAGE-SIZE.

```

## PROCESS-REPORT.

IF LINE-COUNT > PAGE-SIZE  
 PERFORM PROCESS-HEADERS.  
 PERFORM MOVE-DATA.  
 PERFORM ACCUMULATE-TOTALS.  
 PERFORM DISTRIBUTION.  
 PERFORM WRITE-RECORD.  
 PERFORM TOTAL-LINES.  
 PERFORM READ-RECORD.

## PROCESS-HEADERS.

MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.  
 WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.  
 WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM HEADER-LINE-2 AFTER ADVANCING 1.  
 ADD 1 TO PAGE-COUNT.  
 MOVE ZEROES TO LINE-COUNT.

## PRINT-TOTALS.

MOVE TOTAL-BALANCE TO VENDOR-BALANCE-TL.  
 MOVE TOTAL-YTD-PURCHASES TO VENDOR-YTD-PURCHASES-TL.  
 MOVE TOTAL-YTD-PAYMENTS TO VENDOR-YTD-PAYMENTS-TL.  
 WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM TOTAL-LINE AFTER ADVANCING 1.

## PRINT-PRECENT.

COMPUTE PRECENT-OVER-TL = OVER-500/TOTAL-BALANCE \* 100.  
 COMPUTE PRECENT-UNDER-TL = UNDER-500/TOTAL-BALANCE \* 100.  
 MOVE OVER-500 TO OVER-500-TL.  
 MOVE UNDER-500 TO UNDER-500-TL.  
 WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM DISTRIBUTION-HEADER-1  
 AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM DISTRIBUTION-HEADER-2  
 AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM LINE-SPACES AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM DISTRIBUTION-UNDER  
 AFTER ADVANCING 1.  
 WRITE PRINT-RECORD FROM DISTRIBUTION-OVER  
 AFTER ADVANCING 1.

## WRITE-RECORD.

WRITE PRINT-RECORD FROM DETAIL-LINE AFTER ADVANCING 1.

## ACCUMULATE-TOTALS.

ADD VENDOR-BALANCE TO TOTAL-BALANCE.  
 ADD VENDOR-YTD-PURCHASES TO TOTAL-YTD-PURCHASES.  
 ADD VENDOR-YTD-PAYMENTS TO TOTAL-YTD-PAYMENTS.

## MOVE-DATA.

MOVE VENDOR-NUMBER TO VENDOR-NUMBER-DL.  
 MOVE VENDOR-NAME TO VENDOR-NAME-DL.  
 MOVE VENDOR-ADDRESS TO VENDOR-ADDRESS-DL.  
 MOVE VENDOR-BALANCE TO VENDOR-BALANCE-DL.  
 MOVE VENDOR-YTD-PURCHASES TO VENDOR-YTD-PURCHASES-DL.  
 MOVE VENDOR-YTD-PAYMENTS TO VENDOR-YTD-PAYMENTS-DL.

## DISTRIBUTION.

IF VENDOR-BALANCE >= 500  
 ADD VENDOR-BALANCE TO OVER-500  
 ELSE  
 ADD VENDOR-BALANCE TO TOTAL-BALANCE  
 ADD VENDOR-BALANCE TO UNDER-500.

## TOTAL-LINES.

ADD 1 TO LINE-COUNT.

## READ-RECORD.

READ INPUT-FILE INTO VENDOR-INFORMATION  
 AT END MOVE "STOP" TO FLAG.

OPEN-FILES.  
OPEN INPUT INPUT-FILE.  
OPEN OUTPUT PRINT-FILE.

CLOSE-FILES.  
CLOSE INPUT-FILE  
PRINT-FILE.

**APPENDIX I****BASIC PROGRAM DEBUGGING TEST****DIRECTIONS:**

This part of the exam requires you to locate and correct various programming logic errors in a BASIC program. At the beginning of this debugging test, your instructor will distribute a floppy disk containing programs and data files, program documentation and an answer sheet to you. The floppy disk will contain:

1. the BASIC source code for each program, and
2. an input data file to be used by the program.

Each program will be accompanied by the following documentation:

1. a printed copy of the program,
2. a printed copy of the input data file,
3. a description of the program requirements,
4. a description of the program logic error,
5. the current, incorrect printed outputs of the program, and
6. the required, correct printed outputs of the program.

Each of the BASIC source programs will contain one or more logic errors. Each logic error will cause the program to produce incorrect outputs, or results. None of the programs will contain any syntax or execution errors. None of the programs will contain a misspelled variable name. All programs will execute, but will produce incorrect results.

For each program, you are to locate each program logic error and write a description of the cause of the error on your answer sheet. Then you are required to use the computer to edit and execute the program until the program will produce the desired outputs, or results. Your test grade will be based upon your ability:

1. to locate and correctly **describe the logic error** on your answer sheet, and
2. to correct the logic error and **successfully execute** the program from your disk to produce the correct results.

You may use and write on any of the printed documentation provided with your test. You also may use the computer to help you locate and find the error. An answer sheet will be provided so that you may describe the cause of program logic error and to list the debugging tools that you used to find and/or correct the error.

At the beginning of the test you will be given the program documentation and a disk containing all of the test programs and data files. When you complete the debugging requirements for each program, hold up your hand and a test administrator will collect your answer sheet. At that point, you may continue working on the next program. You may only work on one program at a time. Once you hand in an answer sheet, you may NOT edit or change a previous test program on your disk! Any program that is edited or changed after the answer sheet has been handed into the test administrator will receive zero points. However, if you are stuck on a particular program, you are encouraged to proceed to the next test program.

You will be limited to a maximum of one hour and twenty minutes to take the exam. At the end of the exam the instructor will collect your disk and any remaining answer sheets and program documentation.

**PROGRAM 1**                      Program Name: PROG1.BAS

**PROGRAM REQUIREMENTS**

This program will input a vendor data file, which contains the vendor's name, background information, current balance and Y.T.D. information. A traditional report should be prepared, which should include report headers, detail lines and total lines. Financial totals for all vendors should be accumulated for the current balance, Y.T.D Purchases and Y.T.D. Payments and a final total line should be printed at the end of the report.

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The report totals printed on the total line are incorrect. You are to find the location of the logic error that causes the incorrect report totals and correct the program so that the correct report totals are provided.

There is only one logic error in this program.

The complete program listing follows.

Data file for Program 1: VENDORB.DAT

## Record Layout

Field Name	Data Type
Vendor Name	String
Vendor Address	String
Current Balance	String
YTD Purchases	String
YTD Payments	String

"Standish","23","BOX 13455 New York, NY 23157","345.60","100.,""197.84"  
 "MacMillian","19","745 8th Ave Alberta, NM 63562","67.98","4500.,""89.33"  
 "J. Smith","934","1 Barnes St. Pgh, PA 15234","9.60",".89","2340."  
 "Dollars.,""9452","23 Fast Blvd. Sands, CA 65357","889.02","144.56","89.03"

Correct Program Output

VENDOR REPORT					PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PURCH	YTD PMTS	BALANCE
23	Standish	BOX 13455 New York, NY 23157	100.00	197.84	345.60
19	MacMillian	745 8th Ave Alberta, NM 63562	4,500.00	89.33	67.98
934	J. Smith	1 Barnes ST. Pgh, PA 15234	.89	2,340.00	9.60
9452	Dollars	23 Fast Blvd. Sands, CA 65357	144.56	89.03	889.02
VENDOR REPORT TOTALS			4,745.45	2,716.20	1,312.20

Incorrect Program Output

VENDOR REPORT					PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PURCH	YTD PMTS	BALANCE
23	Standish	BOX 13455 New York, NY 23157	100.00	197.84	345.60
19	MacMillian	745 8th Ave Alberta, NM 63562	4,500.00	89.33	67.98
934	J. Smith	1 Barnes ST. Pgh, PA 15234	.89	2,340.00	9.60
9452	Dollars	23 Fast Blvd. Sands, CA 65357	144.56	89.03	889.02
VENDOR REPORT TOTALS			144.56	89.03	889.02



## SOURCE PROGRAM 1

```

DECLARE SUB COUNTLINES (LINE.COUNT!)
DECLARE SUB DOHEADERS (LINE.COUNT!, PAGE.COUNT!, PAGE.SIZE!)
DECLARE SUB OPENIT ()
DECLARE SUB READREC (VENDOR.NAMES$, VENDOR.NOS$,
                    VENDOR.ADDRESS$, VENDOR.BALANCES$,
                    YTD.PURCHASES$, YTD.PAYMENTS$$)
DECLARE SUB ACCUMULATE (VENDOR.BALANCES$, YTD.PURCHASES$,
                       YTD.PAYMENTS$, TOTAL.BALANCE!,
                       TOTAL.PAYMENTS!, TOTAL.PURCHASES!)
DECLARE SUB DETAILLINE (VENDOR.NAMES$, VENDOR.NOS$,
                       VENDOR.ADDRESS$, VENDOR.BALANCES$,
                       YTD.PURCHASES$, YTD.PAYMENTS$$)
DECLARE SUB FINALTOTAL (TOTAL.BALANCE!, TOTAL.PAYMENTS!,
                       TOTAL.PURCHASES!)
DECLARE SUB CLOSEIT ()

' TEST QUESTION : PROGRAM 1
LINE.COUNT = 999: PAGE.COUNT = 1: PAGE.SIZE = 20
TOTAL.BALANCE = 0: TOTAL.PURCHASES = 0: TOTAL.PAYMENTS = 0
CALL OPENIT
DO WHILE NOT EOF(1)
  IF LINE.COUNT > PAGE.SIZE THEN
    CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  END IF
  CALL READREC (VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$,
               YTD.PURCHASES$, YTD.PAYMENTS$$)
  CALL DETAILLINE (VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$,
                  YTD.PURCHASES$, YTD.PAYMENTS$$)
  CALL COUNTLINES (LINE.COUNT)
LOOP
  CALL ACCUMULATE(VENDOR.BALANCES$, YTD.PURCHASES$, YTD.PAYMENTS$,
                 TOTAL.BALANCE, TOTAL.PAYMENTS, TOTAL.PURCHASES)
  CALL FINALTOTAL(TOTAL.BALANCE, TOTAL.PAYMENTS, TOTAL.PURCHASES)
  CALL CLOSEIT
END

SUB ACCUMULATE (VENDOR.BALANCES$, YTD.PURCHASES$, YTD.PAYMENTS$, TOTAL.BALANCE,
               TOTAL.PAYMENTS, TOTAL.PURCHASES)
  TOTAL.BALANCE = TOTAL.BALANCE + VAL(VENDOR.BALANCES$)
  TOTAL.PAYMENTS = TOTAL.PAYMENTS + VAL(YTD.PAYMENTS$$)
  TOTAL.PURCHASES = TOTAL.PURCHASES + VAL(YTD.PURCHASES$$)
END SUB

SUB CLOSEIT
  CLOSE (1)
END SUB

```

```

SUB COUNTLINES (LINE.COUNT)
  LINE.COUNT = LINE.COUNT + 1
END SUB

```

```

SUB DETAILLINE (VENDOR.NAME$, VENDOR.NOS, VENDOR.ADDRESS$, VENDOR.BALANCES,
  YTD.PURCHASES$, YTD.PAYMENTS$)

```

```

  FLINE1$ = "### \ \ \ \ ###,###.## ###,###.## ###,###.##"
  PRINT USING FLINE1$; VAL(VENDOR.NOS), VENDOR.NAME$, VENDOR.ADDRESS$,
    VAL(YTD.PURCHASES$), VAL(YTD.PAYMENTS$), VAL(VENDOR.BALANCES$)
END SUB

```

```

SUB DOHEADERS (LINE.COUNT, PAGE.COUNT, PAGE.SIZE)

```

```

  CLS
  PRINT TAB(30); "VENDOR REPORT"; TAB(72); "PAGE :"; PAGE.COUNT
  PRINT
  PRINT "VEN#"; TAB(6); "VENDOR NAME"; TAB(20); "ADDRESS"; TAB(50);
  PRINT "YTD PURCH"; TAB(62); "YTD PMTS"; TAB(74); "BALANCE"
  PRINT "_____"; TAB(6); "_____"; TAB(20); STRING$(28, "_"); TAB(50);
  PRINT "_____"; TAB(62); "_____"; TAB(74); "_____"
  LINE.COUNT = 0
  PAGE.COUNT = PAGE.COUNT + 1
END SUB

```

```

SUB FINALTOTAL (TOTAL.BALANCE, TOTAL.PAYMENTS, TOTAL.PURCHASES)

```

```

  FLINE1$ =
  "      VENDOR REPORT TOTALS      ###,###.## ###,###.## ###,###.##"
  PRINT
  PRINT USING FLINE1$; TOTAL.PURCHASES, TOTAL.PAYMENTS, TOTAL.BALANCE
END SUB

```

```

SUB OPENIT
OPEN "VENDORB.DAT" FOR INPUT AS #1
END SUB

```

```

SUB READREC (VENDOR.NAME$, VENDOR.NOS, VENDOR.ADDRESS$, VENDOR.BALANCES,
  YTD.PURCHASES$, YTD.PAYMENTS$)
  INPUT #1, VENDOR.NAME$, VENDOR.NOS, VENDOR.ADDRESS$, VENDOR.BALANCES,
    YTD.PURCHASES$, YTD.PAYMENTS$
END SUB

```

**PROGRAM 2**Program Name: PROG2.BAS**PROGRAM DESCRIPTION**

This program will input a student data file, which contains the student's name, credits earned to date and the total quality points earned to date. A traditional report should be prepared, which should include report headers, detail lines and total lines reporting the average student Q.P.A. Each detail line should report the individual student's Q.P.A. An individual student's

Q.P.A. is calculated by the following formula:

$$\text{STUDENT Q.P.A.} = \frac{\text{STUDENT QUALITY POINTS}}{\text{STUDENT CREDITS TO DATE}}$$

Total student quality points and total student credits to date should be accumulated for all students and the average student Q.P.A. should be calculated by using the following formula and reported on the final total line.

$$\text{AVERAGE STUDENT Q.P.A.} = \frac{\text{TOTAL STUDENT QUALITY POINTS}}{\text{TOTAL STUDENT CREDITS TO DATE}}$$

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The individual student Q.P.A. is incorrect, but the average student Q.P.A. is correct. You are to find the location of the logic error that causes the incorrect student Q.P.A. and correct the program so that the correct report totals are provided.

There is only one logic error in this program.

The complete program listing follows.

Input Data file for Program 2: STUDENTB.DAT

## Record Layout

Field Name	Data Type
Student Name	String
Student Number	String
Quality Points	String
Total Credits	String

"Mary Jones", "14", "40", "10"  
 "Doug Smith", "21", "30", "15"  
 "Adam Upp", "20", "20", "12"  
 "Bart Simpson", "7", "30", "10"

Correct Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS EARNED	QUALITY POINTS	STUDENT QPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
20	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00
AVERAGE STUDENT QPA				2.55

Incorrect Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS EARNED	QUALITY POINTS	STUDENT QPA
14	Mary Jones	10	40	0.00
21	Doug Smith	15	30	4.00
20	Adam Upp	12	20	2.00
7	Bart Simpson	10	30	1.66
AVERAGE STUDENT QPA				2.55

## SOURCE PROGRAM TWO

```

DECLARE SUB COUNTLINES (LINE.COUNT!)
DECLARE SUB DOHEADERS (LINE.COUNT!, PAGE.COUNT!, PAGE.SIZE!)
DECLARE SUB OPENIT ( )
DECLARE SUB READREC (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$,
CREDITS.TO.DATES$)
DECLARE SUB ACCUMULATE (QUALITY.POINTS$, CREDITS.TO.DATES$, TOTAL.Q.POINTS!,
TOTAL.CREDITS!)
DECLARE SUB DETAILLINE (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$,
CREDITS.TO.DATES$, GPA!)
DECLARE SUB FINALTOTAL (TOTAL.Q.POINTS!, TOTAL.CREDITS!)
DECLARE SUB CLOSEIT ( )
DECLARE SUB CALCULATEGPA (QUALITY.POINTS$, CREDITS.TO.DATES$, GPA!)

' TEST QUESTION : PROGRAM 2
LINE.COUNT = 999: PAGE.COUNT = 1: PAGE.SIZE = 20
TOTAL.Q.POINTS = 0: TOTAL.CREDITS = 0
CALL OPENIT
DO WHILE NOT EOF(1)
  IF LINE.COUNT > PAGE.SIZE THEN
    CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  END IF
  CALL READREC (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$)
  CALL ACCUMULATE (QUALITY.POINTS$, CREDITS.TO.DATES$, TOTAL.Q.POINTS,
TOTAL.CREDITS)
  CALL DETAILLINE (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$,
CREDITS.TO.DATES$, GPA)
  CALL CALCULATEGPA (QUALITY.POINTS$, CREDITS.TO.DATES$, GPA)
  CALL COUNTLINES (LINE.COUNT)
LOOP
CALL FINALTOTAL (TOTAL.Q.POINTS, TOTAL.CREDITS)
CALL CLOSEIT
END

SUB ACCUMULATE (QUALITY.POINTS$, CREDITS.TO.DATES$, TOTAL.Q.POINTS,
TOTAL.CREDITS)

TOTAL.Q.POINTS = TOTAL.Q.POINTS + VAL(QUALITY.POINTS$)
TOTAL.CREDITS = TOTAL.CREDITS + VAL(CREDITS.TO.DATES$)

END SUB

SUB CALCULATEGPA (QUALITY.POINTS$, CREDITS.TO.DATES$, GPA)

GPA = VAL(QUALITY.POINTS$) / VAL(CREDITS.TO.DATES$)

END SUB

SUB CLOSEIT
CLOSE (1)
END SUB

```

```
SUB COUNTLINES (LINE.COUNT)
  LINE.COUNT = LINE.COUNT + 1
END SUB

SUB DETAILLINE (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$,
  QPA)

  FLINE1$ = "#### \      \      ###      ##      #.##"
  PRINT USING FLINE1$; VAL(STUDENT.NOS$), STUDENT.NAMES$, VAL(QUALITY.POINTS$),
    VAL(CREDITS.TO.DATES$), QPA
END SUB

SUB DOHEADERS (LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  CLS
  PRINT TAB(30); "STUDENT QPA REPORT"; TAB(70); "PAGE :"; PAGE.COUNT
  PRINT
  PRINT "STUD.#"; TAB(10); "STUDENT NAME"; TAB(35); "CREDITS EARNED"; TAB(50);
  PRINT "QUALITY POINTS"; TAB(66); "STUDENT Q.P.A."
  PRINT "_____"; TAB(10); "_____"; TAB(35); "_____"; TAB(50);
  PRINT "_____"; TAB(66); "_____";
  LINE.COUNT = 0
  PAGE.COUNT = PAGE.COUNT + 1
END SUB

SUB FINALTOTAL (TOTAL.Q.POINTS, TOTAL.CREDITS)

  AVERAGE.QPA = TOTAL.Q.POINTS / TOTAL.CREDITS
  FLINE1$ = "        AVERAGE QPA                ###.##"
  PRINT
  PRINT USING FLINE1$; AVERAGE.QPA

END SUB

SUB OPENIT
  OPEN "STUDENTB.DAT" FOR INPUT AS #1
END SUB

SUB READREC (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$)
  INPUT #1, STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$
END SUB
```

**PROGRAM 3****Program Name: PROG3.BAS****PROGRAM DESCRIPTION**

This program will input a student data file, which contains the student's name, credits earned to date and the total quality points earned to date. A high/low analysis report should be prepared, which should include report headers, detail lines and final report lines reporting the name of the student and their Q.P.A., who received either the highest or the lowest Q.P.A. Each detail line should report the individual student's Q.P.A. An individual student's Q.P.A. is calculated by the following formula:

$$\text{STUDENT Q.P.A.} = \frac{\text{STUDENT QUALITY POINTS}}{\text{STUDENT CREDITS TO DATE}}$$

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The analysis lines provided at the end of the student report, which lists the student with the highest and lowest Q.P.A., are incorrect. You are to find the location of the logic error that causes the program to report the incorrect highest Q.P.A. and lowest Q.P.A.. You are to correct the program so that the correct highest and lowest Q.P.A. with corresponding student's name are provided.

There is only one logic error in this program.

The complete program listing follows.

Input Data file for Program 3: STUDENTB.DAT

## Record Layout

Field Name	Data Type
Student Name	String
Student Number	String
Quality Points	String
Total Credits	String

"Mary Jones", "14", "40", "10"  
 "Doug Smith", "21", "30", "15"  
 "Adam Upp", "20", "20", "12"  
 "Bart Simpson", "7", "30", "10"

Correct Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS EARNED	QUALITY POINTS	STUDENT QPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
20	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00

Mary Jones HAS THE HIGHEST QPA OF 4.00  
 Adam Upp HAS THE LOWEST QPA OF 1.67

Incorrect Program Output

STUDENT QPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS EARNED	QUALITY POINTS	STUDENT QPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
20	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00

Bart Simpson HAS THE HIGHEST QPA OF 3.00  
 Adam Upp HAS THE LOWEST QPA OF 3.00



## SOURCE PROGRAM 3

```

DECLARE SUB FINDHI (QPAI, HIGH.QPAI, HIGH.STUDENTS$, STUDENT.NAMES$)
DECLARE SUB FINDLOW (QPAI, LOW.QPAI, LOW.STUDENTS$, STUDENT.NAMES$)
DECLARE SUB PRINTHIGH (HIGH.QPAI, HIGH.STUDENTS$)
DECLARE SUB PRINTLOW (LOW.QPAI, LOW.STUDENTS$)
DECLARE SUB COUNTLINES (LINE.COUNT!)
DECLARE SUB DOHEADERS (LINE.COUNT!, PAGE.COUNT!, PAGE.SIZE!)
DECLARE SUB OPENIT ()
DECLARE SUB READREC (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$)
DECLARE SUB DETAILLINE (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$,
QPAI)
DECLARE SUB FINALTOTAL (TOTAL.Q.POINTS!, TOTAL.CREDITS!)
DECLARE SUB CLOSEIT ()
DECLARE SUB CALCULATEQPA (QUALITY.POINTS$, CREDITS.TO.DATES$, QPAI)

' TEST QUESTION : PROGRAM 3
LINE.COUNT = 999: PAGE.COUNT = 1: PAGE.SIZE = 20
HIGH.QPA = 0: LOW.QPA = 999

CALL OPENIT
DO WHILE NOT EOF(1)
  IF LINE.COUNT > PAGE.SIZE THEN
    CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  END IF
  CALL READREC(STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$)
  CALL CALCULATEQPA(QUALITY.POINTS$, CREDITS.TO.DATES$, QPA)
  CALL DETAILLINE(STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$, QPA)
  CALL COUNTLINES(LINE.COUNT)
LOOP
  CALL FINDHI(QPA, HIGH.QPA, HIGH.STUDENTS$, STUDENT.NAMES$)
  CALL FINDLOW(QPA, LOW.QPA, LOW.STUDENTS$, STUDENT.NAMES$)
  CALL PRINTHIGH(HIGH.QPA, HIGH.STUDENTS$)
  CALL PRINTLOW(LOW.QPA, LOW.STUDENTS$)
  CALL CLOSEIT
END

SUB CALCULATEQPA (QUALITY.POINTS$, CREDITS.TO.DATES$, QPA)
  QPA = VAL(QUALITY.POINTS$) / VAL(CREDITS.TO.DATES$)
END SUB

SUB CLOSEIT
  CLOSE (1)
END SUB

SUB COUNTLINES (LINE.COUNT)
  LINE.COUNT = LINE.COUNT + 1
END SUB

SUB DETAILLINE (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$, QPA)
  FLINE$ = "#### \ \ ### ## #.##"
  PRINT USING FLINE$: VAL(STUDENT.NOS$), STUDENT.NAMES$, VAL(QUALITY.POINTS$),
  VAL(CREDITS.TO.DATES$), QPA
END SUB

SUB DOHEADERS (LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  CLS
  PRINT TAB(30); "STUDENT QPA REPORT"; TAB(70); "PAGE :"; PAGE.COUNT
  PRINT
  PRINT "STUD.#"; TAB(10); "STUDENT NAME"; TAB(35); "CREDITS EARNED"; TAB(50);
  PRINT "QUALITY POINTS"; TAB(66); "STUDENT Q.P.A."
  PRINT "_____"; TAB(10); "_____"; TAB(35); "_____"; TAB(50);
  PRINT "_____"; TAB(66); "_____";
  LINE.COUNT = 0
  PAGE.COUNT = PAGE.COUNT + 1
END SUB

```

```
SUB FINDHI (QPA, HIGH.QPA, HIGH.STUDENT$, STUDENT.NAMES)
  IF HIGH.QPA < QPA THEN
    HIGH.QPA = QPA
    HIGH.STUDENT$ = STUDENT.NAMES$
  END IF
END SUB

SUB FINDLOW (QPA, LOW.QPA, LOW.STUDENT$, STUDENT.NAMES)
  IF LOW.QPA > QPA THEN
    LOW.QPA = QPA
    LOW.STUDENT$ = STUDENT.NAMES$
  END IF
END SUB

SUB OPENIT
OPEN "STUDENTB.DAT" FOR INPUT AS #1
END SUB

SUB PRINTHIGH (HIGH.QPA, HIGH.STUDENT$)
PRINT
H1$ = " \          \ HAS THE HIGHEST QPA OF #.##"
PRINT USING H1$; HIGH.STUDENT$, HIGH.QPA
END SUB

SUB PRINTLOW (LOW.QPA, LOW.STUDENT$)
L1$ = " \          \ HAS THE LOWEST QPA OF #.##"
PRINT USING L1$; LOW.STUDENT$, LOW.QPA
END SUB

SUB READREC (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$)
INPUT #1, STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$
END SUB
```

**PROGRAM 4**Program Name: PROG4.BAS**PROGRAM DESCRIPTION**

This program will input a student data file, which contains the student's name, credits earned to date and the total quality points earned to date. A high/low analysis report should be prepared, which should include report headers, detail lines and final report lines reporting the name of the student and their Q.P.A., who received either the highest or the lowest Q.P.A. Each detail line should report the individual student's Q.P.A. An individual student's Q.P.A. is calculated by the following formula:

$$\text{STUDENT Q.P.A.} = \frac{\text{STUDENT QUALITY POINTS}}{\text{STUDENT CREDITS TO DATE}}$$

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The analysis lines provided at the end of the student report, which lists the student with the highest and lowest Q.P.A., are incorrect. You are to find the location of the logic error that causes the program to report the incorrect highest Q.P.A. and lowest Q.P.A.. You are to correct the program so that the correct highest and lowest Q.P.A. with corresponding student's name are provided.

There are TWO logic errors in this program.

The complete program listing follows.

Input Data file for Program 4: STUDENTB.DAT

## Record Layout

Field Name	Data Type
Student Name	String
Student Number	String
Quality Points	String
Total Credits	String

"Mary Jones", "14", "40", "10"  
 "Doug Smith", "21", "30", "15"  
 "Adam Upp", "20", "20", "12"  
 "Bart Simpson", "7", "30", "10"

Correct Program Output

STUDENT GPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS EARNED	QUALITY POINTS	STUDENT GPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
20	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00

Mary Jones HAS THE HIGHEST GPA OF 4.00  
 Adam Upp HAS THE LOWEST GPA OF 1.67

Incorrect Program Output

STUDENT GPA REPORT				PAGE: 1
STUD. #	STUDENT NAME	CREDITS EARNED	QUALITY POINTS	STUDENT GPA
14	Mary Jones	10	40	4.00
21	Doug Smith	15	30	2.00
20	Adam Upp	12	20	1.66
7	Bart Simpson	10	30	3.00

Adam Upp HAS THE HIGHEST GPA OF 0.00  
 Adam Upp HAS THE LOWEST GPA OF 0.00



```
SUB FINDHI (QPA, HIGH.QPA, HIGH.STUDENTS$, STUDENT.NAMES$)
  IF HIGH.QPA > QPA THEN
    HIGH.QPA = QPA
    HIGH.STUDENTS$ = STUDENT.NAMES$
  END IF
END SUB

SUB OPENIT
OPEN "STUDENTB.DAT" FOR INPUT AS #1
END SUB

SUB PRINTHIGH (HIGH.QPA, HIGH.STUDENTS$)
PRINT
H1$ = " \          \ HAS THE HIGHEST QPA OF #.##"
PRINT USING H1$; HIGH.STUDENTS$, HIGH.QPA
END SUB

SUB PRINTLOW (LOW.QPA, LOW.STUDENTS$)
L1$ = " \          \ HAS THE LOWEST QPA OF #.##"
PRINT USING L1$; LOW.STUDENTS$, LOW.QPA
END SUB

SUB READREC (STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$)
INPUT #1, STUDENT.NAMES$, STUDENT.NOS$, QUALITY.POINTS$, CREDITS.TO.DATES$
END SUB
```

**PROGRAM 5**Program Name: PROG5.BAS**PROGRAM REQUIREMENTS**

This program will input a vendor data file, which contains the vendor's name, background information, current balance and Y.T.D. information. A traditional report should be prepared, which should include report headers, detail lines and total lines. Financial totals for all vendors should be accumulated for the current balance, Y.T.D Purchases and Y.T.D. Payments and a final total line should be printed at the end of the report. Following the final total line a distribution analysis report should appear, listing the dollar amount and percentage of the vendor's outstanding balance "under 500 dollars" and "over 500 dollars" due.

The input record layout and correct report output are provided on the next page.

**DESCRIPTION OF THE DEBUGGING PROBLEM**

On the next page, the current report output is provided. The total balance printed on the total line are incorrect. In addition, the results of the distribution analysis are incorrect. You are to find the location of the logic errors that causes the incorrect total balance and distribution report and correct the program so that the correct distribution report and total balances are provided.

There is only one logic error in this program.

The complete program listing follows.

**Data file for Program 5: VENDORB.DAT****Record Layout**

Field Name	Data Type
Vendor Name	String
Vendor Address	String
Current Balance	String
YTD Purchases	String
YTD Payments	String

"Standish", "23", "BOX 13455 New York, NY 23157", "345.60", "100.", "197.84"  
 "MacMillian", "19", "745 8th Ave Alberta, NM 63562", "67.98", "4500.", "89.33"  
 "J. Smith", "934", "1 Barnes St. Pgh, PA 15234", "9.60", ".89", "2340."  
 "Dollars.", "9452", "23 Fast Blvd. Sands, CA 65357", "889.02", "144.56", "89.03"

**Correct Program Output**

VENDOR REPORT						PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PURCH	YTD PMTS	BALANCE	
23	Standish	BOX 13455 New York, NY 23157	100.00	197.84	345.60	
19	MacMillian	745 8th Ave Alberta, NM 63562	4,500.00	89.33	67.98	
934	J. Smith	1 Barnes ST. Pgh, PA 15234	.89	2,340.00	9.60	
9452	Dollars	23 Fast Blvd. Sands, CA 65357	144.56	89.03	889.02	
VENDOR REPORT TOTALS			4,745.45	2,716.20	1,312.20	

DISTRIBUTION ANALYSIS REPORT			
	DOLLARS	PERCENT	
UNDER 500 DOLLARS BALANCE	423.18	32.25%	
OVER 500 DOLLARS BALANCE	889.02	67.75%	

**Incorrect Program Output**

VENDOR REPORT						PAGE: 1
VEN.#	VENDOR NAME	ADDRESS	YTD PURCH	YTD PMTS	BALANCE	
23	Standish	BOX 13455 New York, NY 23157	100.00	197.84	345.60	
19	MacMillian	745 8th Ave Alberta, NM 63562	4,500.00	89.33	67.98	
934	J. Smith	1 Barnes ST. Pgh, PA 15234	.89	2,340.00	9.60	
9452	Dollars	23 Fast Blvd. Sands, CA 65357	144.56	89.03	889.02	
VENDOR REPORT TOTALS			4,745.45	2,716.99	1,735.38	

DISTRIBUTION ANALYSIS REPORT			
	DOLLARS	PERCENT	
UNDER 500 DOLLARS BALANCE	423.18	24.39%	
OVER 500 DOLLARS BALANCE	889.02	51.23%	



## SOURCE PROGRAM 5

```

DECLARE SUB DISTRIBUTION (VENDOR.BALANCE$, TOTAL.BALANCE!, UNDER.500!, OVER.500!)
DECLARE SUB PRINTPRECENT (TOTAL.BALANCE!, OVER.500!, UNDER.500!)
DECLARE SUB COUNTLINES (LINE.COUNT!)
DECLARE SUB DOHEADERS (LINE.COUNT!, PAGE.COUNT!, PAGE.SIZE!)
DECLARE SUB OPENIT ()
DECLARE SUB READREC (VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$,
YTD.PURCHASES$, YTD.PAYMENTS$)
DECLARE SUB ACCUMULATE (VENDOR.BALANCES$, YTD.PURCHASES$, YTD.PAYMENTS$, TOTAL.BALANCE!,
TOTAL.PAYMENTS!, TOTAL.PURCHASES!)
DECLARE SUB DETAILLINE (VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$,
YTD.PURCHASES$, YTD.PAYMENTS$)
DECLARE SUB FINALTOTAL (TOTAL.BALANCE!, TOTAL.PAYMENTS!, TOTAL.PURCHASES!)
DECLARE SUB CLOSEIT ()

' TEST QUESTION : PROGRAM 1
LINE.COUNT = 999: PAGE.COUNT = 1: PAGE.SIZE = 20
TOTAL.BALANCE = 0: TOTAL.PURCHASES = 0: TOTAL.PAYMENTS = 0
CALL OPENIT
DO WHILE NOT EOF(1)
  IF LINE.COUNT > PAGE.SIZE THEN
    CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  END IF
  CALL READREC(VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$, YTD.PURCHASES$,
YTD.PAYMENTS$)
  CALL ACCUMULATE(VENDOR.BALANCES$, YTD.PURCHASES$, YTD.PAYMENTS$, TOTAL.BALANCE,
TOTAL.PAYMENTS, TOTAL.PURCHASES)
  CALL DISTRIBUTION(VENDOR.BALANCES$, TOTAL.BALANCE, UNDER.500, OVER.500)
  CALL DETAILLINE(VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$,
YTD.PURCHASES$, YTD.PAYMENTS$)
  CALL COUNTLINES(LINE.COUNT)
LOOP
  CALL FINALTOTAL(TOTAL.BALANCE, TOTAL.PAYMENTS, TOTAL.PURCHASES)
  CALL PRINTPRECENT(TOTAL.BALANCE, OVER.500, UNDER.500)
  CALL CLOSEIT
END

SUB ACCUMULATE (VENDOR.BALANCES$, YTD.PURCHASES$, YTD.PAYMENTS$, TOTAL.BALANCE,
TOTAL.PAYMENTS, TOTAL.PURCHASES)
TOTAL.BALANCE = TOTAL.BALANCE + VAL(VENDOR.BALANCES$)
TOTAL.PAYMENTS = TOTAL.PAYMENTS + VAL(YTD.PAYMENTS$)
TOTAL.PURCHASES = TOTAL.PURCHASES + VAL(YTD.PURCHASES$)
END SUB

SUB CLOSEIT
CLOSE (1)
END SUB

SUB COUNTLINES (LINE.COUNT)
LINE.COUNT = LINE.COUNT + 1
END SUB

SUB DETAILLINE (VENDOR.NAMES$, VENDOR.NOS$, VENDOR.ADDRESS$, VENDOR.BALANCES$, YTD.PURCHASES$,
YTD.PAYMENTS$)
  1234567890123456789012345678901234567890123456789012345678901234567890
  FLINE1$ = "### \ \ \ \ \ ###,###.## ###,###.99"
  ##,###.99"
  PRINT USING FLINE1$: VAL(VENDOR.NOS$), VENDOR.NAMES$, VENDOR.ADDRESS$, VAL(YTD.PURCHASES$),
VAL(YTD.PAYMENTS$), VAL(VENDOR.BALANCES$)
END SUB

SUB DISTRIBUTION (VENDOR.BALANCES$, TOTAL.BALANCE, UNDER.500, OVER.500)

  IF VAL(VENDOR.BALANCES$) >= 500 THEN
    OVER.500 = OVER.500 + VAL(VENDOR.BALANCES$)
  ELSE
    UNDER.500 = UNDER.500 + VAL(VENDOR.BALANCES$)
    TOTAL.BALANCE = TOTAL.BALANCE + VAL(VENDOR.BALANCES$)
  END IF

END SUB

```

```

SUB DOWHEADERS (LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
CLS
PRINT TAB(30); "VENDOR REPORT"; TAB(70); "PAGE :"; PAGE.COUNT
PRINT
PRINT "VEN#"; TAB(6); "VENDOR NAME"; TAB(20); "ADDRESS"; TAB(50);
PRINT "YTD PURCH"; TAB(62); "YTD PMTS"; TAB(74); "BALANCE"
PRINT "_____"; TAB(6); "_____"; TAB(20); STRING$(28, "_"); TAB(50);
PRINT "_____"; TAB(62); "_____"; TAB(74); "_____";
LINE.COUNT = 0
PAGE.COUNT = PAGE.COUNT + 1
END SUB

SUB FINALTOTAL (TOTAL.BALANCE, TOTAL.PAYMENTS, TOTAL.PURCHASES)
FLINE1$ = "          VENDOR REPORT TOTALS          ###,###.## ###,###.99"
PRINT
PRINT USING FLINE1$; TOTAL.PURCHASES, TOTAL.PAYMENTS, TOTAL.BALANCE
END SUB

SUB OPENIT
OPEN "VENDORB.DAT" FOR INPUT AS #1
END SUB

SUB PRINTPERCENT (TOTAL.BALANCE, OVER.500, UNDER.500)
PERCENT.UNDER = UNDER.500 / TOTAL.BALANCE * 100
PERCENT.OVER = OVER.500 / TOTAL.BALANCE * 100

PRINT
PRINT "DISTRIBUTION ANALYSIS REPORT"

PRINT
PRINT "          DOLLARS          PERCENT "
P1$ = " UNDER 500 DOLLARS BALANCE  ###,###.##  ##.## %"
PRINT USING P1$; UNDER.500, PERCENT.UNDER
P1$ = " OVER 500 DOLLARS BALANCE  ###,###.##  ##.## %"
PRINT USING P1$; OVER.500, PERCENT.OVER

END SUB

SUB READREC (VENDOR.NAMES$, VENDOR.NO$, VENDOR.ADDRESS$, VENDOR.BALANCE$, YTD.PURCHASES$,
YTD.PAYMENTS$)
INPUT #1, VENDOR.NAMES$, VENDOR.NO$, VENDOR.ADDRESS$, VENDOR.BALANCE$, YTD.PURCHASES$,
YTD.PAYMENTS$
END SUB

```

## APPENDIX L

Microfocus's One Variable Interactive Program Debugger

```

89 PROCEDURE DIVISION.
90 START-HERE.
91     PERFORM INITIALIZE-VALUES.
92     PERFORM OPEN-FILES.
93     PERFORM READ-RECORD.
94     PERFORM PROCESS-REPORT UNTIL FLAG = "STOP".
95     PERFORM ACCUMULATE-TOTALS.
96     PERFORM PRINT-TOTALS.
97     PERFORM CLOSE-FILES.
98     STOP RUN.
99 INITIALIZE-VALUES.
100    MOVE "GO  " TO FLAG.
101    MOVE ZEROES TO TOTAL-YTD-PURCHASES.
102    MOVE ZEROES TO TOTAL-YTD-PAYMENTS.
103    MOVE ZEROES TO TOTAL-BALANCE.
104    MOVE 1 TO PAGE-COUNT.
105    MOVE 999 TO LINE-COUNT.
106    MOVE 20 TO PAGE-SIZE.
107 PROCESS-HEADERS.
108    MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.
109    WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.
Animate-DEBUG1E-----Level=01-Speed=5Ins-Caps-
Num-Scroll F1=help F2=view F3=align F4=exchange F5=where F6=look-up
F9/F10=word-</> Escape Step Go Zoom next-If Perform Reset Break Env
Query Find Locate Text Do 0-9=speed

```

MicroFocus's Interactive Debugger Command Set

Help screen for...  
Help0218

Animate

Page 2 of 2

F1=help	Display previous screen	nx-If	Execute until next IF
F2=view	Display user screen	Perform	Set executed perform level
F3=align	Set this line to 3	Reset	Reset execution position
F4=exchange	Move to other screen	Brk	Set/unset break-points
F5=where	Find current position	Env	Set execution environ.
F6=look-up	Set entered line to 3	Query	Examines data-item
F9=word-left	Move one word to left	Find	Find next occurrence
F10=word-right	Move one word to right	Locate	Locate declaration of item
Escape	Leave Animator	Text	Set screen separator
Step	Execute one instruction	Do	Execute typed COBOL syntax
Go	Execute slowly	0-9	Set default Go speed
Zoom	Execute at full speed		
Wch	Monitor all variables	on current line	

MicroFocus's Interactive Query of a File

```

142             AT END MOVE "STOP" TO FLAG.
143 OPEN-FILES.
144     OPEN INPUT INPUT-FILE.
145     OPEN OUTPUT PRINT-FILE.
146 CLOSE-FILES.
147     CLOSE INPUT-FILE
148     PRINT-FILE.

```

```

Query:   INPUT-FILE-----Level=02-Speed=5-Ins-
Caps-Num-Scroll
F1=help F2=clear F3=hex F4=monitor      ↑ ↓ =up/down data
F7=containing F8=contained F9=same level ↓
Alt Escape
Open input           Last status 00]

```

MicroFocus's Interactive Query of a Data Item

```

103     MOVE 1 TO PAGE-COUNT.
104     MOVE 999 TO LINE-COUNT.
105     MOVE 20 TO PAGE-SIZE.
106     PROCESS-HEADERS
107     MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.
108     WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PA
109     WRITE PRINT-RECORD FROM HEADER-LINE-2 AFTER ADVANCING 2.
110     ADD 1 TO PAGE-COUNT.
111     MOVE ZEROES TO LINE-COUNT.
112     PRINT-TOTALS.
113     MOVE TOTAL-BALANCE           TO VENDOR-BALANCE-TL.
114     MOVE TOTAL-YTD-PURCHASES    TO VENDOR-YTD-PURCHASES-TL.
115     MOVE TOTAL-YTD-PAYMENTS     TO VENDOR-YTD-PAYMENTS-TL.
116     WRITE PRINT-RECORD FROM TOTAL-LINE AFTER ADVANCING 2.
117     PROCESS-REPORT.
118     IF LINE-COUNT > PAGE-SIZE
119     PERFORM PROCESS-HEADERS.
120     PERFORM MOVE-DATA.
121     PERFORM WRITE-RECORD.
122     PERFORM TOTAL-LINES.
123     PERFORM READ-RECORD.

```

```

Query:   PAGE-SIZE-----Level=01-Speed=5-Ins-
Caps-Num-Scroll
F1=help F2=clear F3=hex F4=monitor ↑ ↓ =up/down data
F7=containing F8=contained F9=same level ↓ Alt Escape
020]

```

MicroFocus's Interactive Query of Multiple Data Items**Before the Execution of a Move Statement**

```

133  MOVE VENDOR-NAME          TO VENDOR-NAME-DL.
134  MOVE VENDOR-ADDRESS      TO VENDOR-ADDRESS-DL.
135  MOVE VENDOR-CITY-STATE-ZIP TO VENDOR-CITY-STATE-ZIP-DL.
136  MOVE VENDOR-BALANCE      TO VENDOR-BALANCE-DL.
137  MOVE VENDOR-YTD-PURCHASES TO VENDOR-YTD-PURCHASES-DL.
138  MOVE VENDOR-YTD-PAYMENTS TO VENDOR-YTD-PAYMENTS-DL.
139  TOTAL-LINES.
140  ADD 1 TO LINE-COUNT.
141  READ-RECORD
142  READ INPUT-FILE INTO VENDOR-INFORMATION
143  AT END MOVE "STOP" TO FLAG
144  OPEN-FILES.
145  OPEN INPUT INPUT-FILE.
146  OPEN OUTPUT PRINT-FILE.
147  CLOSE-FILES.
148  CLOSE INPUT-FILE

      INPUT-FILE
      Open input
      last status 00

```

Animate-DEBUG1E-----Level=03-Speed=5-Ins-Caps-Num-Scroll  
 F1=help F2=view F3=align F4=exchange F5=where F6=look-up  
 Step(Wch) Go Zoom nx-If Perform Reset Brk Env Query Find Locate

**After the Execution of a Move Statement**

```

133  MOVE VENDOR-NAME          TO VENDOR-NAME-DL.
134  MOVE VENDOR-ADDRESS      TO VENDOR-ADDRESS-DL.
135  MOVE VENDOR-CITY-STATE-ZIP TO VENDOR-CITY-STATE-ZIP-DL.
136  MOVE VENDOR-BALANCE      TO VENDOR-BALANCE-DL.
137  MOVE VENDOR-YTD-PURCHASES TO VENDOR-YTD-PURCHASES-DL.
138  MOVE VENDOR-YTD-PAYMENTS TO VENDOR-YTD-PAYMENTS-DL.
139  TOTAL-LINES.
140  ADD 1 TO LINE-COUNT.
141  READ-RECORD
142  READ INPUT-FILE INTO VENDOR-INFORMATION
143  AT END MOVE "STOP" TO FLAG
144  OPEN-FILES.
145  OPEN INPUT INPUT-FILE.
146  OPEN OUTPUT PRINT-FILE.
147  CLOSE-FILES.
148  CLOSE INPUT-FILE

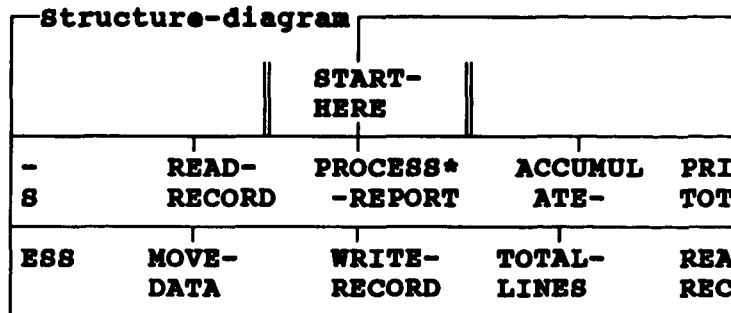
      INPUT-FILE
      Open input
      last status 00

```

Animate-DEBUG1E-----Level=03-Speed=5-Ins-Caps-Num-Scroll  
 F1=help F2=view F3=align F4=exchange F5=where F6=look-up  
 Step(Wch) Go Zoom nx-If Perform Reset Brk Env Query Find Locate

MicroFocus's Structure Chart Interactive Program Debugger

89 PROCEDURE DIVISION.  
 90 START-HERE.  
 91 PERFORM INITIALIZE-VALUES.  
 92 PERFORM OPEN-FILES.  
 93 PERFORM READ-RECORD.  
 94 PERFORM PROCESS-REPORT UN  
 95 PERFORM ACCUMULATE-TOTALS  
 96 PERFORM PRINT-TOTALS.  
 97 PERFORM CLOSE-FILES.  
 98 STOP RUN.



99 INITIALIZE-VALUES.  
 100 MOVE "GO " TO FLAG.  
 101 MOVE ZEROES TO TOTAL-YTD-PURCHASES.  
 102 MOVE ZEROES TO TOTAL-YTD-PAYMENTS.  
 103 MOVE ZEROES TO TOTAL-BALANCE.  
 104 MOVE 1 TO PAGE-COUNT.  
 105 MOVE 999 TO LINE-COUNT.  
 106 MOVE 20 TO PAGE-SIZE.  
 107 PROCESS-HEADERS.  
 108 MOVE PAGE-COUNT TO PAGE-NUMBER-OUT.  
 109 WRITE PRINT-RECORD FROM HEADER-LINE-1 AFTER ADVANCING PAGE.  
 Animate-DEBUG1E-----Level=01-Speed=5Ins-Caps-Num-Scroll  
 F1=help F2=view F3=align F4=exchange F5=where F6=look-up F9/F10=node-  
 </> Escape Step(Wch) Go Zoom nx-If Perform Reset Brk Env Query Find  
 Locate Text Do Alt Ctrl **Structure being recreated**

MicroSoft's Quick Basic Inertactive Debugger

File Edit View Search Run Debug Calls F1=Help

DEBUG1E.BAS

```

DECLARE SUB COUNTLINES (LINE.COUNT!)
DECLARE SUB DOHEADERS (LINE.COUNT!, PAGE.COUNT!, PAGE.SIZE!)
DECLARE SUB OPENIT ()
DECLARE SUB READREC (VENDOR.NAME$, VENDOR.NO$, VENDOR.ADDRESS$,
DECLARE SUB ACCUMULATE (VENDOR.BALANCE$, YTD.PURCHASES$, YTD.PAYMEN
DECLARE SUB DETAILLINE (VENDOR.NAME$, VENDOR.NO$, VENDOR.ADDRESS$,
DECLARE SUB FINALTOTAL (TOTAL.BALANCE!, TOTAL.PAYMENTS!, TOTAL.PU
DECLARE SUB CLOSEIT ()

```

```

' TEST QUESTION : PROGRAM 1
LINE.COUNT = 999: PAGE.COUNT = 1: PAGE.SIZE = 20
TOTAL.BALANCE = 0: TOTAL.PURCHASES = 0: TOTAL.PAYMENTS = 0
CALL OPENIT
DO WHILE NOT EOF(1)
  IF LINE.COUNT > PAGE.SIZE THEN
    CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE)
  END IF
  CALL READREC(VENDOR.NAME$, VENDOR.NO$, VENDOR.ADDRESS$, VENDOR.BA

```

Immediate

Main: DEBUG1E.BAS Context: DEBUG1E.BAS

C 00015:001

Microsoft's Interactive Query and Monitoring of Data Items

File Edit View Search Run Debug Calls F1=Help

DEBUG1E.BAS VENDOR.NAME\$: Standish

READREC VENDOR.NAME\$: &lt;Not watchable&gt;

DEBUG1E.BAS |↑|

```

DECLARE SUB COUNTLINES (LINE.COUNT!)
DECLARE SUB DOHEADERS (LINE.COUNT!, PAGE.COUNT!, PAGE.SIZE!)
DECLARE SUB OPENIT ()
DECLARE SUB READREC (VENDOR.NAME$, VENDOR.NO$, VENDOR.ADDRESS$,
DECLARE SUB ACCUMULATE (VENDOR.BALANCE$, YTD.PURCHASES$, YTD.PAYMEN
DECLARE SUB DETAILLINE (VENDOR.NAME$, VENDOR.NO$, VENDOR.ADDRESS$,
DECLARE SUB FINALTOTAL (TOTAL.BALANCE!, TOTAL.PAYMENTS!, TOTAL.PU
DECLARE SUB CLOSEIT ()

```

' TEST QUESTION : PROGRAM 1

LINE.COUNT = 999: PAGE.COUNT = 1: PAGE.SIZE = 20

TOTAL.BALANCE = 0: TOTAL.PURCHASES = 0: TOTAL.PAYMENTS = 0

CALL OPENIT

DO WHILE NOT EOF(1)

IF LINE.COUNT &gt; PAGE.SIZE THEN

CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE)

END IF

CALL READREC(VENDOR.NAME\$, VENDOR.NO\$, VENDOR.ADDRESS\$, VENDOR.BA

Immediate

Main: DEBUG1E.BAS Context: DEBUG1E.BAS

C 00020:004



Other Microsoft's Quick Basic Debugging Options

File Edit View Search R	Debug	Calls F1=Help
DEBUG1E.BAS VENDOR.NAME\$: S	<b>Add Watch...</b> <b>Watchpoint...</b> Delete Watch. <b>Delete All Watch..</b>	↑ <b>ENDOR.BALANCE</b> <b>MENTS\$,TOTAL.</b> <b>S\$,VENDOR.BAA</b> <b>TURCHASES!</b>
READREC VENDOR.NAME\$: <Not		
DECLARE SUB READREC (VENDOR DECLARE SUB ACCUMULATE (VEN DECLARE SUB DETAILLINE (VOR DECLARE SUB FINALTOTAL (TO DECLARE SUB CLOSEIT ()	<b>Trace On</b> <b>History On</b>	
'TEST QUESTION : PRO LINE.COUNT = 999: PAGE.CN TOTAL.BALANCE = 0: TOTAL.P CALL OPENIT DO WHILE NOT EOF(1) IF LINE.COUNT > PAGE.SIZE THEN CALL DOHEADERS(LINE.COUNT, PAGE.COUNT, PAGE.SIZE) END IF CALL READREC(VENDOR.NAME\$, BENDOR.NO\$, VENDOR.ADDRESS\$, VENDOR.BALANCE\$, YT CALL DETAILLINE(VENDOR.NAME\$, VENDOR.NO\$, VENDOR.ADDRESS\$ VENDOR.BALANCE\$,	<b>Toggle Breakpoint</b> Clear All Breaks Set Next Statement	
Immediate		