TR82-003

Evaluation of a Teaching Approach for Introductory Computer Programming

by

Philip Koltun

A dissertation
submitted to the faculty of
the University of North Carolina at Chapel Hill
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the
Department of Computer Science

Chapel Hill, 1982

[illegible]
Dr. Donald F. Stanat, adviser

David L. Parnas
Dr. David L. Parnas, reader

[illegible]
Dr. Elizabeth Kruesi, reader

(c) Copyright 1982
Philip Koltun

All Rights Reserved

ABSTRACT

PHILIP LOUIS KOLTUN. Evaluation of a Teaching Approach for Introductory Computer Programming. (Under the direction of DONALD F. STANAT.)

An objective evaluation is presented of the applicability of Dijkstra's ideas on program development methodology to the teaching of introductory programming students. The methodology emphasizes development of assertion-based correctness arguments hand-in-hand with the programs themselves and uses a special language to support that approach. Measures of program correctness and programming errors after the first two-thirds of the course indicate that with a batch implementation of the language, the methodology provides a significant advantage over a conventional teaching approach which emphasizes program testing and tracing and uses Pascal on a batch machine. However, that advantage was not maintained when the experimental subjects switched over to Pascal in the latter third of the experiment.

A second set of comparisons demonstrated a significant and impressive advantage with respect to program correctness, programming errors, and time expenditure for students being taught with the conventional approach and using Pascal on a microcomputer over students being taught with the conventional approach and using Pascal on the batch machine. Furthermore, the microcomputer effect was noticeably beneficial for students of marginal ability.

ACKNOWLEDGMENTS

The older one gets the more debts he owes for his intellectual and professional development, so these acknowledgments may stretch a little longer than usual. In my case, I gratefully acknowledge my family for fostering in me a love of learning and a respect for good teaching, and for supporting my efforts to make an original contribution to knowledge in my chosen field. I respect and appreciate the faculties of Carnegie-Mellon University and the University of North Carolina, where I have studied, for setting high examples in pursuing excellence in research and scholarly activities. I thank the faculty and administration of Sangamon State University, where I formerly taught, particularly my dear friend and former colleague, Don Klett, for giving me the opportunity to teach and learn a great deal more computer science that I ever thought possible.

I would like to thank the following members of my committee: Don Stanat, whose stewardship of the DPL project these last several years helped steer a sometimes wayward ship into port, and whose patient counsel helped me, often, along the way; Dave Parnas, the originator of the project of which this dissertation is a part, for setting an example in asking hard questions about challenging, important problems; Betsy Kruesi, whose expertise and encouragement came as a godsend more than once to save this work from foundering; Steve Pizer, whose generous cooperation in carrying out the experiment described herein played such an important role; Tom Wallsten, whose advice on the experimental design and analysis aspects of the work has proven invaluable; and Fred Brooks and Steve Weiss for their insightful comments on this work. Carlo Ghezzi, whose teaching of the DPL course the first time helped me develop my own approach to the subject matter, should also be acknowledged. In addition, I would like to thank Paul Thompson of the Institute for Research in Social Science at UNC, Tom McCabe of McCabe & Associates, and Dave Weiss of the Naval Research Laboratory for freely consulting with me on various aspects of this experiment.

The many students in COMP 14, our introductory programming course, deserve special recognition for their willing participation in efforts to improve undergraduate education. The teaching assistants who helped so much in collecting the data needed by this experiment have also earned my special thanks. Among them were Danny Berrier, Colin Clement, Mario D'Souza, Ernesto Murillo, Manmohan Reddy, and Jane Whang, the graduate teaching assistants, and Ellen Blue, Tricia Carter, Karol Doster, Linda Foglia, and Dawn McKissick, the undergraduate assistants. Not to be forgotten are the people who labored long and hard on the DPL compiler, among them, most recently, Akiko Wakabayashi, Dawn Janney, Deborah Branton, and Marion Varner.

Finally, I acknowledge the many friendly conversations with my officemates Dan Hoffman and Dave Moffat, who share my joy in learning and teaching about computer science.

This work was supported by National Science Foundation grant SED77-18518. At a time in our history when everyone agrees on the critical need for support of science education, yet unprecedented cuts slash away at science education budgets, it is most important of all to acknowledge the Division of Science Education Development and Research's support in sponsoring this project. Deep appreciation is also accorded for National Science Foundation Instructional Equipment grant 1-0-110-3276-XA584, which paid for some of the microcomputers used by subjects in this experiment.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter I

INTRODUCTION

If we are to have a science we must develop more measurement of relevant things. ... I would have more than mere measurement. I would include evaluation, since measurement plus evaluation comes near to judgment which is the ultimate goal.
--R.W. Hamming, 1975[1]

This dissertation has to do, in general, with how to develop computer programs and how to educate programming students. In particular, it has to do with evaluating whether the methodology developed by one of the most respected proponents of structured programming can be applied to teaching beginners. That methodology places its heaviest emphasis on developing proofs of program correctness hand-in-hand with the programs themselves, as a means of establishing from the outset the correctness of the students' endeavors. The method finds its primary intellectual stimulus in the writings of E.W. Dijkstra, particularly _A Discipline of Programming_,[2] which have influenced many programmers, teachers of programming students, and authors of programming texts.

Why is it important, at this date, to evaluate methods of teaching programming? After all, if asked, most professors of computer science would assert that programming students are more effectively taught now than a decade ago, implying that good, or at least better, methods have been found in the interim. Furthermore, many of the best minds in the programming methodology field have moved beyond a discussion of the circumscribed problems encountered in introductory classes, to a consideration of the large-scale undertakings demanded by today's ambitious computer projects, implying that programming-in-the-small has already been mastered.

Several answers occur. First, those professors may well be correct in their assertions, but it would be difficult to substantiate on the basis of statistics. Good teaching, like

---

[1] R.W. Hamming, "A Philosophy for Computer Science or My Prejudices and Confessions," _SIGCSE Bulletin_, Volume 7, Number 4, 1975, pp. 16-18.

[2] E.W. Dijkstra, _A Discipline of Programming_, (Englewood Cliffs: Prentice-Hall, 1976).

good science as Hamming describes it, demands measurement and evaluation. The statistics gathered in the conduct of this project provide a baseline for the state-of-the-teaching-art in 1981, against which programming teaching may be compared in yet another decade.

Second, the gurus of programming methodology may well have mastered their craft, but hundreds of thousands of professional programmers still struggle with theirs, and the statistics presented here will demonstrate that learning to program is still a time-consuming, laborious task. It makes little sense to talk about the engineering of complex systems without also ensuring that students are learning, by the best means available, how to reliably construct the component parts.

The existing body of computer science literature contains the following kinds of writings relevant to the intellectual content of this dissertation:

1. Ideas about how to program, particularly those relating to correctness concerns;

2. Philosophical discussions about how to teach introductory programming;

3. Descriptions of actual teaching experiments;

4. Studies of programming activities including

   a) Data bases of programming project statistics;

   b) Utility of particular techniques (flowcharts, mnemonic variable names, etc.);

   c) Computing environment (machine access);

   d) Language-related effects;

   e) Human factors;

5. Measures of program quality;

6. Methodological problems involved in conducting programming studies using human subjects.

Background literature relevant to the dissertation will be surveyed in the following chapter.

The objectives of the dissertation work include gaining insight into how to program, how to teach programming, how to evaluate the learning of programming, and how to conduct programming experiments involving human subjects. The primary goal is that of objectively evaluating a programming methodology which emphasizes correctness concerns during the development of programs, and utilizes a special programming language to reinforce that concern. A by-product of the study will be a statistical data base that will be useful in assessing the effort involved in learning to program.

The organization of the dissertation provides separate chapters to review the literature, present the experimental hypotheses and design, describe the data collection procedures, analyze the results, and reflect on Dijkstra's notation as an actual programming language.

Chapter II

LITERATURE SURVEY

2.1 PROGRAM CORRECTNESS

If one asserted that programming is taught better now than it was ten years ago, an explanation for the phenomenon might be that the paramount importance of program correctness has finally been recognized. While it was once viewed as acceptable to write a program and then begin its verification and debugging, a realization grew through the 1960's and 1970's of the unacceptability of this strategy of program development. Testing, it was realized, could only show the presence of errors, not their absence.

From Dijkstra's perspective[3] the approach to program correctness up to the mid-60's had treated a program as a mechanism, not unlike a Turing Machine: One tried to prove something about the class of happenings which ensued when one started it in a certain class of initial states. Taking the program as a preexisting entity had proven relatively fruitless in that period. Dijkstra then advocated inverting the process, treating the program as something to be designed, and settling first on what must be proven and what proof techniques could be used, before undertaking the program development.

If one traces the development of structured programming, as Weiner has done,[4] one sees a steady stream of attempts to characterize what it is that language mechanisms accomplish, what it is that programs accomplish, and how one goes about _constructing_ a correct solution to a problem.

---

[3] E.W. Dijkstra, "Correctness Concerns and, Among Other Things, Why They Are Resented," _SIGPLAN Notices_, Volume 10, Number 6, June, 1975, pp.546-550.

[4] L.H. Weiner, "The Roots of Structured Programming," _SIGCSE Bulletin_, Volume 10, Number 1, February, 1978, pp. 243-254.

[5] C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," _Communications of the ACM_, Volume 9, Number 5, 1966, pp.

In 1966, Bohm and Jacopini[5] asserted[6] that any program can be expressed using only sequence (or concatenation), alternation (or selection), and iteration (or repetition) as control mechanisms.

Also in 1966, Naur published a paper[7] in which he described a technique of characterizing conditions that existed at given points in the program text and using those conditions to establish the correctness of the program. That technique evolved into the development of invariant relations. Floyd followed, in 1967, with a paper[8] which used annotated flowcharts that were labeled at the nodes with assertions about the values of program variables at those points, to argue the correct termination of the program. Dijkstra, in 1968, contributed one paper[9] describing the constructive development of a solution to the producer-consumer synchronization problem, and another[10] describing the (provably correct) design of an implemented operating system.

Dijkstra, himself, cites the importance of a 1969 article by C.A.R. Hoare[11] in describing a set of axioms and inference rules to be used in proving program properties. As well as developing the usefulness of invariant relations for proving assertions about repetitive constructs, Hoare's article also influenced thinking about program abstractions, or implementation-independent properties of programs.

That year also saw the circulation of Dijkstra's "Notes on Structured Programming,"[12] in which he cemented the idea

---

366-371.

[6] Later proved, for the class of proper programs, by H. Mills in Mathematical Foundations for Structured Programming, IBM Report FSC 72-6013, 1972.

[7] P. Naur, "Proof of Algorithms by General Snapshots," BIT, Volume 6,4, 1966, pp. 310-316.

[8] R.W. Floyd, "Assigning Meanings to Programs," Proceedings of Symposia in Applied Mathematics," Volume 19, American Mathematics Society, 1967, pp. 19-32.

[9] E.W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," BIT, Volume 8,3, 1968, pp. 174-186.

[10] E.W. Dijkstra, "The Structure of the THE Multiprogramming System," Communications of the ACM, Volume 11, Number 5, 1968, pp. 341-346.

[11] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM, Volume 12, Number 10, 1969, pp. 576-583.

of using enumeration, mathematical induction, and abstraction as reasoning patterns with the process of stepwise decomposition of programs. Dijkstra demonstrated how, as control structures, concatenation and selection may be understood by enumerative reasoning, repetition may be understood by inductive reasoning, and how abstraction may be used to consider what a program action does independently of its implementation (how it works). Thus, he showed that starting at the top level of refinement with a demonstrably correct solution statement, and breaking that statement into subactions whose individual effects could be understood as abstractions and whose combined effect could be understood by either enumerative or inductive reasoning, an iterative decomposition process enables the entire solution to be specified. Of equal importance, the abstraction process which separates the effect of an action from its implementation also separated for the first time the mathematical concern for program correctness from the engineering concern for program efficiency. This attention to a "separation of concerns," as Dijkstra[13] calls it, of two goals historically intermingled, marked a point of major advance in the state of the art of programming.

In spite of the elegance of its presentation, it took some time for Dijkstra's notes to manifest their effect. In 1972, a survey of program correctness still devoted itself largely to a posteriori proofs of program correctness and only minimally to their usefulness in the construction of programs.[14]

At about this time, the work of Harlan Mills at IBM received its due attention. In several papers[15] he presented ideas complementary to those of Dijkstra, in viewing the stepwise decomposition process as one of specifying the program function in terms of lower-level single-entry, single-exit subfunctions, using only

---

[12] Later published in O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, _Structured Programming_, (New York: Academic Press, 1972)

[13] Dijkstra, "Correctness Concerns".

[14] B. Elspas, K.N. Levitt, R.J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," _Computing Surveys_, Volume 4, Number 2, 1972, pp. 97-147.

[15] H.D. Mills, "Top Down Programming in Large Systems," _Debugging Techniques in Large Systems_, B. Rustin (ed.), (Englewood Cliffs: Prentice-Hall, 1971).

Mills, _Mathematical Foundations for Structured Programming_, 1972.

composition, selection, and repetition as control structures. Mills also helped develop the idea of the Chief Programmer Team for attacking large-scale software projects.[16]

The years that followed 1972 saw a consolidation and exploration of the earlier structured programming ideas, and included the development of a language intended to embody those concepts.[17] A highwater mark of sorts was reached in 1975 with the International Conference on Reliable Software,[18] in Los Angeles, where program correctness concerns dominated the discussions. The fullest flower of expression for program correctness ideas came in 1976, with Dijkstra's A Discipline of Programming, and in 1979, with the publication of Structured Programming, by Linger, Mills, and Witt.[19]

By the late 1970's, with great emphasis now being placed on developing formal and machine-aided proofs of program correctness, several authors pleaded to keep proof processes in their proper perspective. DeMillo, Lipton, and Perlis argued[20] that the aspiration of programming methodologists to develop formal mathematical machine-digestible proofs for their programs was a false one on several grounds: Firstly, mathematicians themselves treat the proof process as largely a social one, wherein they try to convince their colleagues of the correctness and utility of the theorems they propose. Computer scientists would be well-advised to regard program proofs in the same manner.[21]

---

[16] F.T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Volume 11, Number 1, 1972, pp. 56-71.

[17] C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica, Volume 2,4, 1973, pp. 335-355.

[18] Proceedings of the International Conference on Reliable Software, Los Angeles, April 21-23, 1975. Also published as SIGPLAN Notices, Volume 10, Number 6, June, 1975.

[19] R.C. Linger, H.D. Mills, and B.I. Witt, Structured Programming, (Reading, Mass.: Addison-Wesley, 1979).

[20] R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," Communications of the ACM, Volume 22, Number 5, 1979, pp. 271-280.

[21] The process of sharing one's programs with one's colleagues and trying to convince them of the programs' correctness has come to be called "structured walkthroughs" and stems from a philosophy called "egoless

Secondly, any proof involves many axioms that often go unstated because they are well-known to the proof's audience. Any machine-verifiable proof would have to include so many axioms that the proof would attain unmanageable length. Thirdly, it would be unlikely that greater confidence would exist in the proof than in the original program itself; to attain such confidence, one would need to have great faith, indeed, in the program-verifying program that produced the proof.

Dijkstra argues for simplicity of program proofs, and suggests that the length of the correctness proof for a program could be accepted as an objective measure of the "elegance" of the program and the suitability of the language constructs it uses.[22]

Mills best states the case for correctness arguments when he notes the profound difference, in a precise mental activity such as programming, between finding even a single error and finding no errors at all. The more errors that are found in the testing and debugging process, the more cause arises for doubting the thought process that developed the program. Thus, he states, the objective should be to write programs that are correct from the start. A proof should be regarded not as an infallible statement of correctness but as a subjective conviction that a given hypothesis leads to a given result. To wit:

> The ultimate faith you can have in a program is in the thought process that created it. With every error you find in testing and use, that faith is undermined. Even if you have found the last error left in your program, you cannot prove it is the last. So your real opportunity to know you have written a correct program is to never find the first error in it, no matter how much it is inspected, tested, and used.[23]

---

programming" developed by Gerry Weinberg, and a practice, developed at IBM. See G. Weinberg, <u>The Psychology of Computer Programming</u>, (New York: Van Nostrand Reinhold, Co., 1971) for the former, and M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," <u>IBM Systems Journal</u>, Volume 15,3, 1976, pp. 182-211, for the latter.

[22] Dijkstra, "Correctness Concerns".

[23] H.D. Mills, "How to Write Correct Programs and Know It," <u>SIGPLAN Notices</u>, Volume 10, Number 6, June, 1975, pp. 363-370.

Perhaps it is this new sense of confidence in the thought processes by which programs are created and communicated that enables some people to maintain that computer programming is taught better today than it was a decade ago. Research work continues in the program correctness area, particularly in the areas of software specifications, development of large-scale software, and design of languages to support data and control abstractions and the development of large programs.[24]

2.2 IDEAS ABOUT HOW TO TEACH PROGRAMMING

The second body of literature to be reviewed expresses ideas about how to teach programming. The literature, particularly the SIGCSE Bulletin of the ACM's Special Interest Group on Computer Science Education, is replete with proposals on how to teach introductory programming. Attempting to review all that has ever been written on the subject will not be tried. However, the 1980 survey by Ulloa[25] provides an excellent overview of much that deserves reading. Only one text has been titled with words suggesting guidance to the reader on how to teach programming.[26] That volume, however, reports the proceedings of a conference dominated more by discussions of programming language design than by pedagogy or programming methodology.

A review of several significant philosophical discussions of programming instruction will follow. Those writings which discuss experiments conducted to evaluate specific instructional techniques will be reserved for a later section.

---

[24] R.B. Anderson, Proving Programs Correct, (New York: John Wiley & Sons, 1979), Chapter 5.

[25] M. Ulloa, "Teaching and Learning Computer Programming: A Survey of Student Problems, Teaching Methods, and Automated Instructional Tools," SIGCSE Bulletin, Volume 12, Number 2, July, 1980, pp. 48-64

[26] W.M. Turski (Ed.), Programming Teaching Techniques, (New York: American Elsevier Publishing Co., 1973).

[27] C.B. Kreitzberg and L. Swanson, "A Cognitive Model for Structuring an Introductory Programming Curriculum," AFIPS Conference Proceedings, Volume 43: 1974 National Computer Conference, (Montvale, NJ: AFIPS Press, 1974), pp. 307-311.

R.E. Mayer, "The Psychology of How Novices Learn Computer

Several attempts have been made[27] to place computer programming instruction in the context of general learning theory. Discussion has centered on the conditions that must exist for meaningful learning, including the need for appropriate models of computation, advance organizers (which provide a brief introduction to new concepts in terms of previously learned ideas), and aids to assist transfer of learning.

Others have tried to summarize what current nationwide practice in introductory programming instruction is or should be. For instance, Hanson and Maly[28] advocate an approach which emphasizes algorithms rather than programming language and teaches a problem-solving methodology whose final stage, only, involves translation of an algorithm into a well-structured program.

Some have developed either augmented or restricted dialects of popular programming languages in an attempt to facilitate development of well-structured programs.[29] Others have developed program design languages to enable the expression of algorithms in structured English.[30]

Schneider attempts[31] to develop a consensus on what the goals of a programming course should be by enumerating ten principles for the course. Those principles include that

1. The starting point in programming is a clear, concise problem statement.

Programming," <u>Computing Surveys</u>, Volume 13, Number 1, 1981, pp. 121-141.

[28] A. Hanson and K. Maly, "A First Course in Computer Science: What It Should Be and Why," <u>SIGCSE Bulletin</u>, Volume 7, Number 1, February, 1975, pp. 95-101.

[29] W.R. Bezanson, "Teaching Structured Programming in FORTRAN with IFTRAN," <u>SIGCSE Bulletin</u>, Volume 7, Number 1, February, 1975, pp. 196-199.

L.P. Meissner and R.L. Hinkins, "B4TRAN: A Structured Mini-Language Approach to the Teaching of FORTRAN," <u>SIGCSE Bulletin</u>, Volume 7, Number 1, February, 1975, pp. 200-205.

[30] T.R. Nanney, "Computer Science: An Essential Course for the Liberal Arts," <u>SIGCSE Bulletin</u>, Volume 8, Number 3, September, 1976, pp. 102-105.

[31] G.M. Schneider, "The Introductory Programming Course in Computer Science -- Ten Principles," <u>SIGCSE Bulletin</u>, Volume 10, Number 1, February, 1978, pp. 107-114.

2. The programming course should emphasize the development of algorithms.

3. The duality of data structures and algorithms in the programming process must be presented.

4. A programming language rich in data and control structures should be presented.

5. Language presentation should concentrate on semantics and program characteristics rather than syntax.

6. Programming style must be emphasized from the beginning.

7. Debugging should be presented as a formal subject.

8. So should program testing and verification.

9. Documentation should also be formally presented.

10. Students should be introduced to real programming applications and real programming environments, including maintenance activities and programmer teams.

A 1979 survey by Lemos[32] reports the results of inquiries to 306 business administration and computer science departments regarding their introductory programming courses. Lemos found that there were ten distinct ways in which instructors tended to organize the introductory course:

1. An emphasis on structured programming;

2. An emphasis on modular programming (how to partition a program into units or "modules");

3. A grammatical approach (in which the syntax of a programming language is presented, construct by construct) or, alternatively, a "whole program" approach (in which whole programs, albeit simple ones, are presented for study, much as a foreign language class based on conversation in whole sentences might operate);

[32] R. Lemos, "Teaching Programming Languages: A Survey of Approaches," SIGCSE Bulletin, Volume 11, Number 1, February, 1979, pp. 174-181.

4. A spiral approach (which presents increasingly complex sample programs that build on each other);

5. A problem analysis approach (which concentrates on the development of language-independent solutions);

6. A computer modeling approach (which emphasizes communicating to the student an appropriate model of computational processes);

7. Computer-assisted instruction;

8. Instructional television;

9. Egoless programming (in which students read and share the use of others' programs);

10. Team programming and debugging techniques.

As Lemos points out, however, while many of these approaches seem intuitively appealing, "they lack any history of empirical evidence attesting to their pedagogical effectiveness."

By the late-1970's only a few authors, still, had turned their attention to the inclusion in introductory courses of material on program correctness through mathematical argument (as opposed to program verification through testing). Among textbook authors, Conway, Gries, and Zimmerman,[33] and Perlis[34] were exceptions, though the latter book was intended for an audience somewhat more mature than introductory programming students. Texts by Wulf, Shaw, Hilfinger, and Flon,[35] and Gries[36] both emphasize a correctness-based approach to programming, but are also aimed at an audience more sophisticated than beginning programmers.

The SIGCSE Bulletin does contain descriptions of several programming courses organized around correctness concerns. Of those, only Gerhart[37] relates experiences in

---

[33] R. Conway, D. Gries, and E.C. Zimmerman, A Primer on Pascal, (Cambridge: Winthrop Publishers, 1976).

[34] A.J. Perlis, Introduction to Computer Science, (New York: Harper & Row, 1975).

[35] W.A. Wulf, M. Shaw, P. Hilfinger, and L. Flon, Fundamental Structures of Computer Science, (Reading, Mass.: Addison-Wesley, 1981).

[36] D. Gries, The Science of Programming, (New York: Springer-Verlag, 1981).

using the approach with an introductory class, and notes of program proving that

> its main role right now is to prevent errors rather than to provide any iron-clad guarantees that programs are correct. The very act of making assertions and attempting a proof elicits numerous assumptions and forces a rigorous check of programs which can often reduce later debugging time, catch subtle errors which would escape detection during testing, and lead to more pointed and useful documentation.

Gerhart instructs her classes to present a prose "argument" that their programs are correct, an argument that should be designed to convince the grader that the program satisfies the assignment. Because proofs themselves can contain errors, she also advocates systematic testing of student programs.

Maurer relates his approach[38] to teaching program correctness in classes designed for students ranging from second-year programming through graduate level. The approach involves assertion verification, primarily as related to run-time conditions that may arise. He gives no indication of using correctness concerns in program development.

Lastly, Jones and Walsh[39] describe plans for teaching a course for advanced undergraduates and graduate students that emphasizes techniques for writing correct programs. Their approach focuses on verifying the consistency between programs and their specifications, on utilizing input and output assertions and invariant relations for characterizing what programs accomplish, and on using top-down refinement and abstract data structures for developing program structure. Their approach to developing correct programs comes closest to approximating the one used in the introductory programming class described later in this dissertation.

---

[37] S.L. Gerhart, "Methods for Teaching Program Verification," SIGCSE Bulletin, Volume 7, Number 1, February, 1975, pp. 172-178.

[38] W.D. Maurer, "The Teaching of Program Correctness," SIGCSE Bulletin, Volume 9, Number 1, February, 1977, pp. 142-144.

[39] G.A. Jones and A.M. Walsh, "A Course in Program Verification for Programmers," SIGCSE Bulletin, Volume 10, Number 1, February, 1978, pp. 213-216.

Of final interest, several academics relate their experiences in administering introductory programming courses without lectures. Bowles[40] describes a course at the University of California at San Diego, organized around computer-assisted instruction on microcomputers. Software developed for the course includes automated quiz programs and a bookkeeping and class scheduling system. Student programming problems emphasize graphics and string manipulation. Daly, Embley, and Nagy[41] state that "Although it is easy to say _what_ students should learn in CS237 (or any other introductory programming course), it is difficult to say _how_ they should learn it." The authors observe that students seem to learn best from direct computer feedback on programs submitted for execution, from carefully worked-out examples, and from one-to-one assistance, and seem to learn less in a traditional lecture setting than might be expected.

2.3 EXPERIMENTAL EVALUATIONS OF TEACHING APPROACHES

Though programming experiments often use students as subjects because of their easy availability, those studies which primarily investigate mode of computer usage or language feature utility will be dealt with later, even if they use students as subjects. This section will be reserved for discussing controlled evaluation of teaching methods used in introductory programming courses. While many authors misuse the term "experiment," as in "experimental course," to refer only to something that may or may not work, a reasonable standard of experimental control, in the usual scientific sense, will be a characteristic of the studies reviewed here. That standard alone, apart from the meaningfulness of the results, removes from consideration a large portion of studies in the computer science education literature.

Among the studies of how to teach introductory programming are those attempting to predict which students will do well in the beginning programming course. Typical of these efforts is Newsted[42] in which two regression

[40] K. Bowles, "A CS1 Course Based on Stand-Alone Microcomputers," _SIGCSE Bulletin_, Volume 10, Number 1, February, 1978, pp. 125-127.

[41] C. Daly, D. Embley, and G. Nagy, "A Progress Report on Teaching Programming to Business Students Without Lectures," _SIGCSE Bulletin_, Volume 11, Number 1, February, 1979, pp. 247-251.

equations were used with a number of variables to predict final course grade and end-of-semester student self-perception of ability. College GPA, prior programming experience, and career orientation to the computer field were found to be positive predictors; (greater) time spent on the course, and (high incidence of) working in groups proved to be negative predictors. From the negative predictors Newsted concludes that "though poorer students may spend much time and ask many questions of their instructors and fellow students, it won't improve their grade. If they are going to learn at all, they can do it on their own as well as in a large lecture course with discussion sections." He sees these predictors as support for a program of individualized instruction in programming.

Petersen and Howe[43] likewise studied predictors of academic success in introductory courses and concluded that only college grade point average and general intelligence contributed significantly to their regression model.

One might conclude from these studies that students who do well in general will likely succeed in computer programming as well. Another possible conclusion might hold that, inasmuch as only 60% of the variance in course grade was explained in each study by the variables used in the regression equations, further attempts at prediction might be warranted to locate other predictor variables. Weinberg's stress on work habits and personality factors involved in programming,[44] in particular, suggests that more than test score-type variables might be involved in learning how to program.

In that regard, Cheney[45] explores the possibility that cognitive style (the problem-solving methodology employed by an individual in a decision situation) could predict a person's programming ability. Cheney compares analytic problem solvers (those who use a structured approach to decision making, seek underlying causal relationships, and try choosing optimal alternatives) to heuristic problem solvers (those who use intuition, common sense, and trial-and-error methods with feedback for selecting alternatives).

---

[42] P.H. Newsted, "Grade and Ability Predictions in an Introductory Programming Course," SIGCSE Bulletin, Volume 7, Number 2, June, 1975, pp. 87-91.

[43] C.G. Petersen and T.G. Howe, "Predicting Academic Success in Introduction to Computers," AEDS Journal, Fall, 1979, pp. 182-191.

[44] Weinberg, The Psychology of Computer Programming.

[45] P. Cheney, "Cognitive Style and Student Programming Ability: An Investigation," AEDS Journal, Summer, 1980, pp. 285-291.

He concludes that analytic decision makers tend to perform better on programming exams than heuristic problem solvers. The validity of his results may be compromised, however, by the reader's observation that the instructional methods and examinations used for measurement favored the analytic types. An important pedagogical question, one which has been little explored to date, is whether individualized instruction can be developed to match individual learning styles.[46]

Attention is now turned from attempts to predict academic success toward attempts to assess the utility of particular teaching approaches. In an early and tentative study, Lucas and Kaplan[47] examine the effect of forcing students to write structured (goto-less) programs, concluding that assignments involving program maintenance were easier for their experimental group than for their control (unstructured) group, and that students using structured programming techniques displayed greater improvement in attitude and performance, as time went on, than did the control group.

In a series of studies, Lemos[48] has explored the value of peer review and team debugging activities in an introductory COBOL programming course. In his most extensive study, the experimental group's lectures were supplemented by in-class reading and critiquing of program listings for each of five homework problems, while a control group received only additional lecture material. Randomly selected three-person teams were formed in the experimental

---

[46] See, for instance, DiMarco, Bird, and Norton, "Life Style, Learning Style, Learning Structure, Their Congruences and Student Attitudes and Performance in a Data Processing Course," _Journal of Educational Data Processing_, Volume 16, Number 2, 1979, pp. 1-8.

[47] H.C. Lucas and R.B. Kaplan, "A Structured Programming Experiment," _Computer Journal_, Volume 19,2, 1976, pp. 136-138.

[48] R.S. Lemos, "A Comparative Study of the Effectiveness of Team Interaction in COBOL Programming Language Learning," (Ph.D Dissertation, UCLA, 1977), _Dissertation Abstracts International_, Volume 38, 1977, pp. 2269B-2270B.

________., "An Implementation of Structured Walk-Throughs in Teaching COBOL Programming," _Communications of the ACM_, Volume 22, Number 6, 1979, pp. 335-340.

________., "Structured Walk-Throughs and Student Ratings of Faculty Effectiveness Versus Expediency," _Journal of Educational Data Processing_, Volume 16, Number 1, 1979, pp. 1-8.

group, and for each assignment class members were expected to turn in program flowcharts, listings, two critiques (done by classmates) of their first listing, a summary of all errors detected on other team members' listings, an error analysis for each run attempt, and the final run results. A comparison of scores on a common final exam testing knowledge of language rules, ability to read and debug a program, and ability to write a program revealed that the experimental group performed significantly better in actually writing COBOL programs in an exam situation than did the control group. Furthermore, the experimental group used fewer runs in completing their homework problems and showed no significant difference in the number of homework problems completed from the control group whose subjects worked independently.

Curiously, however, Lemos found that in evaluating the instructor's effectiveness, the control group rated the instructor significantly higher on five of 12 measures (command of subject, clarity of expression, availability to students, desire to teach, and enthusiasm for subject matter) than did the group which used the structured walk-throughs.

In a related study,[49] Lemos' work investigated different ways of assessing student proficiency in programming language learning, and indicated a direct relationship between the ability to read programs and the ability to write programs. He views this result as very important since evaluation of reading ability takes significantly less time than evaluation of writing ability.

Among investigations performed by other academics, plans were made to assess the relative merits of breadth and depth in introductory computer courses.[50] Perhaps uniquely among all the studies reported in the literature, Stoddard, Sedlmeyer, and Lee planned to evaluate the effects of two parallel first-year courses of study with measurements taken during a common second year of study in an undergraduate data processing curriculum. Exposure to three different programming languages (FORTRAN, BASIC, and RPG) was to be compared with deeper exploration of algorithm development in just one language.

---

[49] R.S. Lemos, "Measuring Programming Language Proficiency," AEDS Journal, Summer, 1980, pp. 261-273.

[50] S.D. Stoddard, R.L. Sedlmeyer, and R.G. Lee, "Breadth or Depth in Introductory Computer Courses: A Controlled Experiment." SIGCSE Bulletin, Volume 11, Number 1, February, 1979, pp. 41-44.

[51] P. Hsia and F.E. Petry, "A Framework for Discipline in Programming," IEEE Transactions on Software Engineering,

Finally, Hsia and Petry[51] report on an introductory programming course experiment emphasizing a disciplined, engineering-like approach to program development. For the experimental group, the programming process was broken down into stages of problem analysis, solution design, test planning, peer review, coding and compilation, testing, and acceptance. Test cases were designed before the coding process was begun. The control group used a conventional approach involving flowcharting, coding, testing and debugging, and documentation. All students were required to keep time and run logs for each of three problems, and to copy their final source programs onto a system tape for subsequent testing on the instructor's data. Time logs revealed only a modest increase in effort (16% or about two hours more per assignment) for the experimental group. Analysis of errors from final runs on composite test data showed the disciplined group's programs to be significantly more error-free (81% to 67% one semester in which the experiment was tried, 85% to 65% the next semester) than the conventional group's. However, the methodology was not foolproof: On one problem, only 43% of the disciplined group (20% of the conventional group) achieved error-free solutions.

2.4 PROGRAMMING STUDIES

The broad general category of programming studies will be broken down into five subcategories: studies that contribute a data base on some aspect of programming activity; studies of particular programming techniques or tools such as flowcharting; studies evaluating different modes of computer usage, such as timesharing and batch processing; studies focusing on programming languages, either taken as a whole or taken feature by feature; and studies focusing on human factors in the programming process.

Weinberg[52] has been a source of inspiration for over a decade to researchers in this area and provides a font of ideas for further investigation concerning psychological dimensions of programming activity. Shneiderman[53] provides a comprehensive summary of research into human factors in computer systems and lists numerous suggestions for further

---

Volume SE-6, Number 2, March, 1980, pp. 226-232.

[52] Weinberg, The Psychology of Computer Programming.

[53] B. Shneiderman, Software Psychology, (Cambridge, Mass.: Winthrop Publishers, 1980).

research. A recent survey by Sheil[54] also supplies a useful summary of activity in this area. Brooks' entertaining book[55] includes statistics on numerous large-scale development efforts.

2.4.1 Statistical Summaries of Programming Phenomena

Studies of how people use actual programming languages, of what kinds of programming errors people make, and of how programmers engage in testing and debugging activities stand out in this area. A landmark study by Knuth[56] drew samples of programs from academic and industrial programming environments and compiled comprehensive statistics on language structures used in actual FORTRAN programs. That work inspired several similar studies involving other languages, among them that of Elshoff,[57] who examined program size, readability, and complexity in a commercial environment.

Youngs studied error-proneness in programming[58] as the subject of his dissertation and in subsequent work, and published useful error data summaries, including relative error proneness of individual language features, for a small sample of programs taken from a programming class.

Nagy and Pennebaker[59] devised an automated system for capturing student programs and comparing them for changes to previous runs. Their study revealed that 80% of the follow-up runs involved changes to only a single statement.

---

[54] B.A. Sheil, "The Psychological Study of Programming," _Computing Surveys_, Volume 13, Number 1, 1981, pp. 101-120.

[55] F.P. Brooks, Jr., _The Mythical Man-Month_, (Reading, Mass.: Addison-Wesley, 1975).

[56] D.E. Knuth, "An Empirical Study of FORTRAN Programs," _Software--Practice & Experience_, Volume 1, Number 2, 1971, pp. 105-133.

[57] J.L. Elshoff, "An Analysis of Some Commercial PL/I Programs," _IEEE Transactions on Software Engineering_, Volume SE-2, Number 2, 1976, pp. 113-120.

[58] E.A. Youngs, "Human Errors in Programming," _International Journal of Man-Machine Studies_, Volume 6,4, 1974, pp. 361-376.

[59] G. Nagy and M.C. Pennebaker, "Automatic Analysis of Student Programming Errors," _International Journal of Man-Machine Studies_, Volume 6, 1974, pp. 563-578.

Further, their data led them to believe that "each new mistake is discovered only once a previous mistake has been corrected."

A later study by Litecky and Davis[60] collected statistics on error occurrence in student COBOL programs. They found that 20% of the possible error types accounted for 80% of the errors, but that only four of the eighteen high-frequency errors were "error prone", that is, traceable to anomalies in the language's design, itself. Of additional significance, they found that over 80% of the compiler's error diagnoses were inaccurate, an unfortunate occurrence for a beginning programming course.

Typically, studies such as the above have sought to provide guidance to language designers, compiler designers, and/or programming language instructors as to how languages are actually used by programmers.

As far as program debugging goes, little concrete work has been accomplished. In addition to the previously cited work reporting error counts, Gould and Drongowski[61] reported that assignment statement errors were the most difficult to unravel in their study, and Gould[62] found that debugging was more efficient on programs the subjects had debugged previously (although with different bugs). Myers[63] reported that in a study of professional programmers debugging a small PL/1 program, the most cost-efficient strategy consisted of two programmers independently looking for errors and combining their results. Sheppard et al[64] reported that minor variations in the structured control mechanisms used in programs did not significantly affect the ease of debugging. Gannon and Horning[65] present statistics

---

[60] C. Litecky and G.B. Davis, "A Study of Errors, Error Proneness and Error Diagnosis in COBOL," <u>Communications of the ACM</u>, Volume 19, Number 1, 1976, pp. 33-37.

[61] J.D. Gould and P. Drongowski, "An Exploratory Study of Computer Program Debugging," <u>Human Factors</u>, Volume 16,3, 1974, pp. 258-277.

[62] J.D. Gould, "Some Psychological Evidence on How People Debug Computer Programs," <u>International Journal of Man-Machine Studies</u>, Volume 7, Number 2, 1975, pp. 151-182.

[63] G.J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," <u>Communications of the ACM</u>, Volume 21, Number 9, 1978, pp. 760-768.

[64] S.B. Sheppard, B. Curtis, P. Milliman, and T. Love, "Modern Coding Practices and Programmer Performance," <u>Computer</u>, December, 1979, pp. 41-49.

[65] J.D. Gannon and J.J. Horning, "Language Design for

on error persistence in the context of a discussion on language feature selection for reliable software design. Hetzel[66] has investigated different program verification strategies in a tightly controlled experimental setup, reporting that specification testing and selective program testing were equally more effective than program reading as a means of program verification. While the above generally study errors and debugging at the small-program level, Weiss has investigated error analysis on large-scale projects.[67]

In general, it might be stated that the scarcity of useful research into debugging as a psychological activity might be cited as one more reason to develop programming methodologies in which errors are never permitted to occur.

Love's dissertation[68] relating human information processing abilities to programming performance contains many useful statistics on computer usage and program attributes for an introductory programming course.

2.4.2 Programming Techniques

Among the programming techniques or practices that have received the most attention is that of flowcharting. While flowcharting of program designs was a popular practice in the earlier days of programming and proved useful in non-programming activities,[69] research has either found flowcharting to be of no significant advantage compared to other techniques,[70] or to be inferior to a program design

---

Programming Reliability," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, 1975, pp. 179-191.

[66] W.C. Hetzel, "An Experimental Analysis of Program Verification Methods," (Ph.D Dissertation, University of North Carolina, 1976), Dissertation Abstracts International, Volume 37, 1977, p. 4054B.

[67] D.M. Weiss, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," The Journal of Systems and Software, Volume 1, 1979, pp. 57-70.

[68] L.T. Love, "Relating Individual Differences in Computer Programming Performance to Human Information Processing Abilities," (Ph.D Dissertation, University of Washington, 1977), Dissertation Abstracts International, Volume 38, 1977, p. 1443B

[69] R. Kammann, "The Comprehensibility of Printed Instructions and Flowchart Alternative," Human Factors, Volume 17,2, 1975, pp. 183-191.

language (PDL) for expressing program designs.[71]

Sheppard, Kruesi, and Curtis[72] studied the effects of symbology (including natural language, a constrained program design language, and flowchart symbols) and spatial arrangements (sequential, branching, and hierarchical) on the comprehension of software specifications and found that forward- and backward- tracing questions were answered more quickly from specifications presented in PDL or flowchart symbols than in natural language.

Indentation of programs has generally been regarded as advantagecus to comprehension. However, none of the reported studies support that contention.[73]

Likewise, mnemonic variable names have long been held to be valuable in aiding program comprehension. Experimental attempts to support that hypothesis have met with mixed results, however. Sheppard et al[74] found that different mnemonic levels of variable names had no significant effect in a comprehension experiment. Newsted[75] reported that groups using nonmnemonic names outperformed mnemonic groups on program comprehension tasks. Shneiderman reports,[76] however, that mnemonic names aided program comprehension.

Finally, with respect to the expected benefits of program commenting, the experimental results are not as convincing as one would hope. Shneiderman[77] found programs with global-level comments to be significantly easier to modify than programs lacking such comments. However,

---

[70] B. Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," <u>Communications of the ACM</u>, Volume 20,6, 1977, pp. 373-381.

[71] H.R. Ramsey, M.E. Atwood, and J.R. Van Doren, <u>A Comparative Study of Flowcharts and Program Design Languages for the Detailed Procedural Specification of Computer Programs</u> (Denver: Science Applications, Inc. 1978).

[72] S.B. Sheppard, E. Kruesi, and B. Curtis, "The Effects of Symbolcgy and Spatial Arrangement on the Comprehension of Software Specifications," <u>Proceedings of the Sixth International Conference on Software Engineering</u>, (New York: IEEE Press, 1981), pp. 207-214.

[73] Shneiderman, <u>Software Psychology</u>, pp.72-74.

[74] S.B. Sheppard, B. Curtis, P. Milliman, and T. Love, "Modern Coding Practices and Programmer Performance," <u>Computer</u>, Volume 12, Number 12, 1979, pp. 41-49.

[75] P.R. Newsted, "FORTRAN Program Comprehension as a

Sheppard, in comparing programs containing either global or in-line comments with programs lacking such comments, was unable to find a significant difference in performance on program modification tasks.[78]

While primarily exploring human factors in software development, Basili and Reiter[79] developed empirical evidence to support the contention that programmer teams using a disciplined methodology for software development have an advantage over either individuals or teams using an ad hoc methodology, in terms of average development costs, average number of errors encountered during implementation, and control flow complexity of the program product.

2.4.3 Mode of Computer Usage

A significant line of experiments has explored the effect that mode of computer access has on programmer productivity or the ability of students to learn to program. Sackman's book,[80] Man-Computer Problem Solving, provides the most comprehensive discussion, including experiments done at the U.S. Air Force Academy involving students. That work was preceded by an earlier Sackman study[81] which has been more often cited for its statistical evidence of huge individual variability in programmer performance. The earlier investigation dealt primarily with conditions for

---

Function of Documentation," School of Business Administration, University of Wisconsin, Milwaukee, undated.

[76] Shneiderman, Software Psychology, pp. 70-72.

[77] B. Shneiderman, "Measuring Computer Program Quality and Comprehension," International Journal of Man-Machine Studies, Volume 9, 1977, pp. 465-478.

[78] Sheppard, Curtis, Milliman, and Love, "Modern Coding Practices"

[79] V. Basili and R.W. Reiter, Jr., "An Investigation of Human Factors in Software Development," Computer, Volume 12,12, December, 1979, pp. 21-38.

[80] H. Sackman, Man-Computer Problem Solving, (Princeton: Auerbach Publishers, 1970).

[81] H. Sackman, W.J. Erikson, and E.F. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," Communications of the ACM, Volume 11, Number 1, 1968, pp. 3-11.

successful program debugging, and indicated an advantage to online activity. Observations about individual variability were added as an afterthought.

Comparisons of time-sharing and batch processing systems as to effectiveness for supporting introductory programming coursework have proven inconclusive. An early study by Smith[82] concluded that instant turnaround (simulated time-sharing) was superior as measured by elapsed time from first run to last and ratio of number of runs to number of trips to the computation center (higher ratio viewed as better). Skelton,[83] however, concluded that no statistically significant difference on either the Problem Solving Ability or the FORTRAN Programming Ability tests was found that was attributable to the mode of computer access.

2.4.4 <u>Programming Language Studies</u>

Yet another area of programming studies involves investigating the utility of individual language features or of languages as whole entities for producing desirably-structured programs. Furuta and Kemp[84] provide a good general survey of the subject.

Very little has been attempted in the way of evaluating or comparing, on a rigorous basis, whole languages for teaching introductory programming. In fact, very few attempts have been made to evaluate whole languages for any purpose. Notable among those efforts are the works of Reisner and of Ledgard, Whiteside, Seymour, and Singer.

Reisner[85] advocates making psychological testing part of the design and development process for new languages. She did just that, evaluating SEQUEL, a relational data base

---

[82] L.E. Smith, "A Comparison of Batch Processing and Instant Turnaround," <u>Communications of the ACM</u>, Volume 10, Number 8, 1967, pp. 495-500.

[83] J.E. Skelton, "Time-Sharing Versus Batch Processing and Teaching Beginning Computer Programming: An Experiment," <u>AEDS Journal</u>, March, 1972, pp.91-97, and June, 1972, pp. 103-109.

[84] R. Furuta and P.M. Kemp, "Experimental Evaluation of Programming Language Features: Implications for Introductory Programming Languages," <u>SIGCSE Bulletin</u>, Volume 11, Number 1, 1979, pp. 18-21.

[85] P. Reisner, "Use of Psychological Experimentation as an Aid to Development of a Query Language," <u>IEEE Transactions on Software Engineering</u>, Volume SE-3, Number 3, 1977, pp. 218-229.

language under development, in relationship to SQUARE, a preexisting data base language. Using both programmers and non-programmers, she investigated overall learnability of the new language, learnability of individual features of the languages, and types and frequencies of errors made.

Ledgard et al[86] attempted to decide whether English language commands or notational commands were more useful for commercial text editors. Evaluating the work of inexperienced, familiar, and experienced users on a 20-minute editing task after training on one of two editors, they concluded that the use of commands resembling English phrases resulted in far better performance. Subjects "could not conceive of editing power or function as something different from the appearance of the actual commands. This suggests that language designers must be as much concerned with surface syntax as with functional features if they mean to design a product to optimize user performance."

As far as empirical testing of individual language features goes, Gould[87] concluded, in the context of a study on how people debug programs, that errors in assignment statements were harder to detect than array or iteration bugs.

Sime, Green, and Guest[88] examined conditional statements, particularly as to the utility of including sequence information (specifying the order in which statements are executed) and taxon information (describing the conditions under which a given action is performed). They note that production systems normally present sequence information, but leave taxon information up to the human reader to discover; decision tables normally present the taxon information, but leave the sequence information up to the human reader to discover. Redundant information in a conditional "else" branch, as Dijkstra advocates,[89] adds taxon information. Nesting of conditionals (as opposed to using goto's) adds sequence information. Sime, Green, and

---

[86] H.F. Ledgard, J.A. Whiteside, W. Seymour, and A. Singer, "An Experiment on Human Engineering of Interactive Software," <u>IEEE Transactions on Software Engineering</u>, Volume SE-6, Number 6, 1980, pp. 602-604.

[87] Gould, "Some Psychological Evidence on How People Debug Computer Programs".

[88] M.E. Sime, T.R.G. Green, and D.J. Guest, "Scope Marking in Computer Conditionals -- A Psychological Evaluation," <u>International Journal of Man-Machine Studies</u>, Volume 9, Number 1, 1977, pp. 107-118.

[89] E.W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," <u>Communications of the ACM</u>, Volume 18, Number 8, 1975, pp. 453-457.

Guest found in their studies that nested redundant conditionals were very effective language structures, particularly in debugging, which requires taxon information as well as sequence information. (Debugging typically requires answers to the questions "If certain conditions are met, what actions will be taken?" and "If a certain action evidently was taken, what conditions must have existed?")

Gannon has done the most extensive work in empirical evaluation of language features as guidance to language designers. In one comprehensive study,[90] he evaluated type, frequency, and persistence of errors made by students in using two similar languages which differed only in several carefully controlled ways. Those ways included order of operator precedence; expression orientation versus statement orientation; form of logical connectives; use of semicolon as either separator or terminator; inclusion or exclusion of a case statement; form of repetition statement; bracketing of compound statements or expressions; scope rules; and inclusion or exclusion of named constants. Gannon's results indicate support for the use of the semicolon as a statement terminator and for requiring the explicit inheritance of global variables when so desired, but no support for a strict right-to-left evaluation of (all-equal-precedence) operators as is found in APL.

In a later study,[91] Gannon reports evidence for concluding that static data typing reduces errors in an least one environment, when compared with languages that permit typing determinable only from statement usage.

Finally, Weinberg, in The Psychology of Computer Programming, lists a number of desirable attributes of programming languages and gives suggestions as to their empirical evaluation. Those attributes include uniformity or consistency of structure; compactness, relative to the psychological concept of "chunking",[92] with more program information or power per chunk being desirable; locality, wherein all parts of a program relevant to a particular concern are found in the same place; linearity of executable statements, arguing for minimization of explicit program branching; and non-error-proneness, wherein inherent psychological ambiguity of program structures is minimized.

---

[90] Gannon and Horning, "Language Design for Programming Reliability".

[91] J.D. Gannon, "An Experimental Evaluation of Data Type Conventions," Communications of the ACM, Volume 20, Number 8, 1977, pp. 584-595.

[92] G.A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," Psychological Review, Volume 63, 1956, pp. 81-97.

2.4.5 Human Factors

Most of the studies that one might group under the heading of "human factors" have already been mentioned elsewhere, notably programming team organization, mode of computer access, and study, a la Weinberg, of personality factors as they are involved in the programming process. Psychological complexity of the programming process will be dealt with later, under the heading of measures of program complexity. Individual variability of programming subjects will be mentioned later under methodological considerations for programming experiments.

This section will consider the cognitive dimensions of the programming process. Mayer[93] and Miller[94] provide useful summaries of concerns in this area.

Several models have been proposed for examining the performance of human users of computer systems, among them the work of Card, Moran, and Newell[95] and Embley and Nagy[96] on modeling text editor usage. Brooks[97] has extensively used protocols (spoken revelations of thought patterns) of programmers working on problems, in order to model coding behavior. He proposes that a programmer is always in one of three distinct states of behavior: understanding, method-finding, or coding, with the method-finding activity being independent of a particular programming language. His model proposes a production system with coding "rules" to explain programmer behavior, and explains differences in programmer performance in terms of differential possession of rules.[98]

---

[93] Mayer, "The Psychology of How Novices Learn Computer Programming".

[94] L.A. Miller, Behavioral Studies of the Programming Process, National Technical Information Service Report #AD/A-061-633, October, 1978.

[95] S.K. Card, T.P. Moran, and A. Newell, "The Keystroke-Level Model for User Performance Time with Interactive Systems," Communications of the ACM, Volume 23, Number 7, 1980, pp. 396-410.

[96] D.W. Embley and G. Nagy, "Behavioral Aspects of Text Editors," ACM Computing Surveys, Volume 13, Number 1, 1981, pp. 33-70.

[97] R. Brooks, "Toward a Theory of the Cognitive Processes in Computer Programming," International Journal of Man-Machine Studies, Volume 9, 1977, pp. 737-751.

[98] See also A. Newell and H.A. Simon, Human Problem Solving, (Englewood Cliffs: Prentice-Hall, 1972), on protocols.

In a similar vein, Larkin, McDermott, Simon, and Simon[99] have compared expert and novice performance in solving physics problems in a way that may also be applicable to studying programmer behavior. While considerable knowledge obviously constitutes a prerequisite to expert skill, "recognition of a pattern often evokes from memory stored information about actions and strategies that may be appropriate in contexts in which the pattern is present" and that may be useful in guiding development of a problem's interpretation and solution. "This capacity to use pattern-indexed schemata is probably a large part of what we call physical intuition."

Shneiderman has performed an experiment[100] which supports a view of information processing differences between the novice and expert programmer. He examined the abilities of subjects to memorize two sequences of FORTRAN statements, one a proper executable program, the other consisting of valid statements in scrambled order. While all subjects did poorly in recalling the scrambled sequence, the more experienced programmers performed significantly better on the actual program, suggesting a chunking effect[101] in which more program content per chunk may be retained in short-term memory by the expert programmer than by the novice.

Love has more deeply explored the relationship of information processing abilities to individual differences in programming performance for his doctoral dissertation work.[102] His objective was to determine whether introductory

---

[99] J. Larkin, J. McDermott, D.P. Simon, and H.A. Simon, "Expert and Novice Performance in Solving Physics Problems," Science, Volume 208, Number 20, June, 1980, pp. 1335-208.

[100] B. Shneiderman, "Exploratory Experiments in Programmer Behavior," International Journal of Computer and Information Science, Volume 5, Number 2, 1976, pp. 123-143.

[101] Each human is assumed to have a capacity to store a similar number of "chunks" of information in short-term memory, though the size or content of chunks may differ across individuals. See H.A. Simon, "How Big Is a Chunk?" Science, Volume 183, 1974, pp. 482-488.

See also A.D. de Groot, Thought and Choice in Chess, (New York: Basic Books, Inc., 1965), for an experiment similar to Shneiderman's that involved recall of actual and scrambled game board situations by master and novice chess players.

[102] Love, "Relating Individual Differences in Computer

programming performance was related to the ability to process information quickly and accurately, and his method used four measures of information processing capability (recall of assigned variable values, recall of serial digits, perceptual speed in comparing strings of digits, and subjective organization of words in a free-recall learning task) and several measures of programming performance (including number of runs needed to complete the assigned task and frequency of program changes across successive runs). Love observed that students who performed well on the variable value recall task, as well as those who performed well on the serial digit recall task, took fewer runs to complete their programming assignments. Students who performed better on the free-recall task reported fewer logical errors in their programs. However, students who did well in remembering variable values also took longer to locate errors in their programs, a counter-intuitive result. As Love states, "Altogether we have evidence here for a relationship between programming performance and human information processing ability, albeit complex!"

Lastly, in this area, some comments might be made under the heading of decision-making under uncertainty. Programming tasks are commonly assigned with a complete set of technical specifications, but with no statement whatever of which performance goals (number of runs, elapsed time, program size, program efficiency, etc.) to optimize. Under such conditions, each subject of an experiment may choose to optimize his own individual goal. Weinberg[103] demonstrated the power of this phenomenon. Five groups of experienced programmers were given the same programming task, each group being given a separate performance goal to optimize. Then the groups were rated on all the goals. Each group outperformed the others on its own individual goal, exhibiting an ability to trade off one performance attribute for another. Whether this ability extends to introductory programming students is open to speculation. However, this phenomenon may explain some of the variability in individual programmer performance observed in some studies, such as that of Sackman, Erikson, and Grant,[104] where no particular performance goals were reported as being given the programmer subjects.

Programming Performance to Human Information Processing Abilities".

[103] G. Weinberg, "The Psychology of Improved Programming Performance," Datamation, November, 1972, pp. 82-85.

[104] Sackman, Erikson, and Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance."

2.5 MEASURES OF PROGRAM COMPLEXITY

If one talks about methodologies for programming or methodologies for teaching programming, there must also be some way of evaluating the quality of the resultant program products. Computational complexity analyses have focused on the executional efficiency of the program algorithm.[105] Analyzing psychological complexity of the resultant program provides another means of assessment. Attempts have been made to identify an intrinsic relationship between program properties and programmer performance on a given programming task, for instance, reading or debugging programs. Because of the dominant role that maintenance activities play in the software life cycle, program complexity measures have sought to characterize how difficult a program is for programmers to work with, that is, locate and correct undetected implementation errors and modify program modules to incorporate specification changes.

In the last decade, a number of metrics have been proposed and empirically evaluated. Among them are the works of Halstead,[106] McCabe,[107] Chapin,[108] and Chen.[109] Reviews and comparisons of the metrics are contained in Fitzsimmons and Love[110] and in Baker and Zweben,[111] as well as in other studies. Because of the focus of attention on Halstead's and McCabe's metrics, the discussion will be limited here to their studies.

---

[105] See, for instance, A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, (Reading, Mass: Addison-Wesley Publishing Co., 1974).

[106] M.H. Halstead, Elements of Software Science, (New York: Elsevier North-Holland, 1977).

[107] T.J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Volume SE-2, Number 4, 1976, pp. 308-320.

[108] N. Chapin, "A Measure of Software Complexity," AFIPS Conference Proceedings, Volume 48: 1979 National Computer Conference, (Montvale, NJ: AFIPS Press, 1979), pp. 995-1002.

[109] E.T. Chen, "Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, Volume SE-4, Number 3, 1978, pp. 187-194.

[110] A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," ACM Computing Surveys, Volume 10, Number 1, 1978, pp. 3-18.

In 1972, Halstead began publishing articles about his work, which characterized algorithms and the languages in which they were expressed in an attempt to establish a scientific basis for the study of programs. He focused on the number of distinct operators in an implementation and the total usage of all operators in that implementation, plus the number of distinct operands in an implementation and the total usage of all operands in that implementation. From these units he developed an equation for the expected program length which was shown to correlate very highly with the observed length in a variety of settings. He also developed characterizations for potential volume (the shortest possible expression of an algorithm) and actual volume (which expresses the conciseness of the algorithmic representation in a particular language), and for programming effort.

Empirical studies[112] have shown the predictive value of Halstead's effort metric for the number of bugs that will be discovered in an implementation and for program comprehensibility, as measured by program recall and the ability to debug programs. (The lower the effort metric, the lower the number of bugs that will occur and the higher the program comprehensibility.) This metric estimates the number of mental discriminations needed in implementing a program once the algorithm is known, and has been shown useful in predicting a value for the actual observed time needed to implement the program. Halstead's metrics have also proven useful for quantitatively analyzing technical prose as well as computer programs.

McCabe independently developed a graph-theoretic measure of program complexity that depends only on the decision structure of a program, not its physical size. In essence, his metric characterizes the "structuredness" of a program. It describes the number of basis paths which, when taken in combination, can generate all possible paths through the program. The metric has applicability, therefore, for characterizing the testability as well as the psychological complexity of a program, and could be used for deciding when a program module has become too complex and should be divided into sub-modules. The appeal of the metric, in practice, is that it can be computed very simply as the number of conditions or predicates in a program plus one.

---

[111] A.L. Baker and S. Zweben, "A Comparison of Measures of Control Flow Complexity," _IEEE Transactions on Software Engineering_, Volume SE-6, Number 6, 1980, pp. 506-512.

[112] See Fitzsimmons and Love, "A Review and Evaluation of Software Science."

McCabe's ideas, like Halstead's, have their advocates and some empirical support for their utility. Among the supporters are Myers,[113] who suggests a modified interval metric incorporating both the number of conditions and the number of decisions; Elshoff and Marcotty,[114] who advocate using only the number of decisions; and Walsh,[115] who describes the usefulness, in a large-scale weapons system development project, of using a McCabe metric cutoff value of ten, for determining module size in a complex program. In Myers' words, "Although it is an extremely simple concept, V(G) appears to be a practical complexity measure because it is easy to calculate, it confirms subjective opinions about complexity, and it is consistent with studies showing a high correlation between the number of decisions in a module and the module's complexity and error proneness."

Curtis, Sheppard, and Milliman[116] have investigated the use of software complexity metrics for predicting programmer performance, as measured by the time to locate and correct bugs in three FORTRAN programs. Working with larger-sized programs than were used in their previous study,[117] Halstead's effort metric and McCabe's cyclomatic complexity metric were related to the difficulty programmers experience in locating errors in code, with the stronger relationship established for the Halstead metric. A curvilinear relationship was found for Halstead's effort metric and programmer performance, suggesting that as Halstead's effort

---

[113] G.J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity, <u>SIGPLAN Notices</u>, Volume 12, Number 10, 1977, pp. 61-64.

[114] J.L. Elshoff and M. Marcotty, "On the Use of the Cyclomatic Number to Measure Program Complexity," <u>SIGPLAN Notices</u>, Volume 13, Number 12, 1978, pp. 29-40.

[115] T.J. Walsh, "A Software Reliability Study Using a Complexity Measure," <u>AFIPS Conference Proceedings, Volume 48: 1979 National Computer Conference</u>, (Montvale, NJ: AFIPS Press, 1979), pp. 761-769.

[116] B. Curtis, S.B. Sheppard, and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," <u>Proceedings of the Fourth International Conference on Software Engineering</u>, (New York: IEEE, 1979).

[117] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," <u>IEEE Transactions on Software Engineering</u>, Volume SE-5, Number 2, 1979, pp. 96-104.

metric grows larger "a program becomes more psychologically complex, but the increments in difficulty grow smaller and smaller."

It appears that research on complexity metrics will continue into the foreseeable future,[118] particularly related to keeping programmers and managers aware of their programming product's logical complexity and to helping them estimate the time and effort needed for their coding, testing, and maintenance work.

2.6 METHODOLOGICAL CONSIDERATIONS IN PERFORMING EXPERIMENTS

While computer scientists have been actively performing experiments on human subjects for some fifteen years, only recently has widespread attention been focused on the sufficiency of experimental designs employed. A sense has grown that computer science experiments should be evaluated for methodology with the same rigor as that applied in the behavioral and natural sciences.[119]

Both Brooks[120] and Moher and Schneider[121] have written of the methodological considerations involved in formulating appropriate software experiments, with Moher and Schneider noting, "Although the literature contains numerous references to the use of experimental methods, there are few references on investigations into the methodology itself." Sheil[122] has also written of experimental concerns.

---

[118] J.C. Zolnowski and D.B. Simmons, "Taking the Measure of Program Complexity," National Computer Conference Proceedings, 1981, (Arlington, Va.: AFIPS Press, 1981), pp. 329-336.

G.M. Schneider, R.L. Sedlmeyer, and J. Kearney, "On the Complexity of Measuring Software Complexity," National Computer Conference Proceedings, 1981, (Arlington, Va.: AFIPS Press, 1981), pp. 317-322.

[119] D.L. Parnas, letter titled "Dubiety of Increased Funding for Experimental Computer Science," Communications of the ACM, Volume 24, Number 3, 1981, pp. 162-163.

[120] R. Brooks, "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," Communications of the ACM, Volume 23, Number 4, 1980, pp. 207-213.

[121] T. Moher and G.M. Schneider, "Methodology and Experimental Results in Software Engineering," International Journal of Man-Machine Studies, Volume 16, Number 1, 1982, pp. 65-87.

The issues that have been raised about experimental methodology include the following:

1. Generalizability of results -- Will results obtained with beginning programming students generalize to professional programmers, even when task performance has been seen[123] to vary with programming experience, or, as Weinberg puts it,[124] will the psychology of programming become the "psychology of programmer trainees"? Will results obtained with small-sized programs generalize to large-scale systems, even when scaling up is not just a side issue in software engineering, but the crux of the matter?

2. Selection of subjects -- Given the apparent variability in programmer performance and the cost inherent in conducting research with real programmers, can subject populations be selected that are small, yet representative, and large enough to produce the desired experimental effect? Moher and Schneider[125] argue that a few simple biographical variables, both experiential and aptitudinal, if taken into effect, can reduce the unexplained variability in performance by about 50%. Removing the effects of dependable predictor variables can substantially reduce estimates of variance and therefore result in a reduction in the number of subjects needed for an experiment.

3. Appropriateness of measures -- What are the underlying variables of interest in the experiment and how do they relate to the aspects of performance actually being measured in the experiment? Lemos[126] found a direct relationship between program reading ability and program writing ability. Few others have

---

[122] Sheil, "The Psychological Study of Programming."

[123] For example, Shneiderman, "Exploratory Experiments in Programmer Behavior."

[124] Weinberg, The Psychology of Computer Programming.

[125] T. Moher and G.M. Schneider, "Methods for Improving Experimentation in Software Engineering," Sixth International Conference on Software Engineering, (New York: IEEE Press, 1981), pp. 224-233.

[126] Lemos, "Measuring Programming Language Proficiency."

sought to establish so directly a link between their performance measures and some aspect of programming ability.

4. Magnitude of experimental effect -- Can a sufficiently strong experimental effect be induced, given the error variance typically present in programming experiments? Can satisfactory materials, addressing such matters as performance goals[127] and requirements, be prepared for the experimental treatment?

5. Unobtrusiveness of measures -- Can performance data be collected unobtrusively, as with the program "drain" of Nagy and Pennebaker,[128] or, at the other extreme, will the experimenter induce a so-called "Hawthorne Effect,"[129] in which experimental observation of subjects itself produced changes (improvements) in subject performance?

Among the recent write-ups of experiments which exhibit a high level of awareness of experimental issues or tight experimental control are those of Hetzel,[130] Stoddard, Sedlmeyer, and Lee,[131] and Sheppard, Kruesi, and Curtis.[132] A recent issue of Software Engineering Notes, which includes eight proposals for software experiments and a discussion of design issues to be considered in making proposals, also provides useful guidance on the subject of methodological

[127] See, for instance, Weinberg, "The Psychology of Improved Programming Performance."

[128] Nagy and Pennebaker, "Automatic Analysis of Student Programming Errors."

[129] Named after productivity experiments performed at the Hawthorne Works of the Western Electric Company. See, for instance, K.R. London, The People Side of Systems, (London: McGraw-Hill, 1976), Chapter 3.

[130] Hetzel, "An Experimental Analysis of Program Verification Methods."

[131] Stoddard, Sedlmeyer, and Lee, "Breadth or Depth in Introductory Programming Courses: A Controlled Experiment."

[132] Sheppard, Kruesi, and Curtis, "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications."

issues.[133]

[133] "Proposals for Tool and Methodology Evaluation Experiments," SIGSOFT First Software Engineering Symposium on Tool and Methodology Evaluation, Software Engineering Notes, Volume 7, Number 1, 1982, pp. 6-75.

Chapter III

THE EXPERIMENT

3.1 BACKGROUND

Publication, in 1976, of E.W. Dijkstra's monograph, A Discipline of Programming, was met with widespread, though not universal, acclaim.[134] In the words of one reviewer, himself a distinguished programming methodologist,

> The material represents a tight distillation of ideas over a lifetime of one of the deepest thinkers in programming today... A Discipline of Programming is a landmark in programming methodology. The unity and power of the theoretical ideas will be the basis for many textbooks in explanation and elaboration over the next decade, and for a whole generation of more effective programmers. The work is a rich source of insights, large and small.[135]

However, studying that work has proved to be a challenging task even for advanced computer science graduate students and computing professionals. Though its form is clearly inappropriate for study by introductory students (and was not intended to be so used), its lessons may nevertheless be communicated to any audience.

The central theme of Dijkstra's book is that the arguments necessary to convince oneself of a program's correctness must be developed hand-in-hand with the program itself, and even more strongly, that the necessity to

---

[134] For a critical opinion of Dijkstra's work, see P.W. Abrahams' review in Computing Reviews, Volume 19, Number 5, May, 1978, pp. 177-179. Abrahams states that development of the programs presented in Dijkstra's book depends more on fortuitous insight than application of a consistent methodology. Furthermore, he says, no time or space constraints are given for the problems presented, nor are criteria stated for the tradeoffs evidently applied in the program development process.

[135] H.D. Mills, Computing Reviews, Volume 17, Number 11, November, 1976, pp. 416-418.

develop a correctness argument should guide the program's construction. An outcome of this idea is that one needs a language in which to express the program requirements or specifications, a language in which to express the program itself, and a language with which to reason about what the program accomplishes. Given widespread agreement among scholars in every intellectual discipline that language shapes our thoughts and actions, it should be clear that the three languages or notations needed for programming, as mentioned above, must be chosen with special care.

Dijkstra describes what a program must accomplish in terms of logical assertions taking the form of output predicates or postconditions on the program's data. He regards the program, then, as a predicate transformer, which, when started with some true initial predicate or precondition on the program's data, terminates with the postcondition being established as true, thus "transforming" the state described by the precondition into the state described by the postcondition. A "weakest" precondition may be formulated which describes the least restrictive (most inclusive) set of initial states or conditions for which the program "works" (terminates establishing the required postcondition). Variables are regarded not so much for their role as the object of computations as for their usefulness in describing the state, or progress, of computations. In particular, variables are needed for formulating the precondition and postcondition assertions.

The language used for expressing programs themselves, by implication, must be formulated especially carefully, to admit of only those language structures which lend themselves to a formal definition of semantics and to tight patterns of logical reasoning. Effective reasoning must be the driving concern in shaping the language, not efficient programming. The resultant language, developed by Dijkstra and described in _A Discipline of Programming_, makes no pretense at being a fully-implementable production language. It is, instead, intended to be a mechanism for communicating algorithms and enabling author and reader to reason together about programs.

Among the language's novel features are a guarded command structure for both alternative and repetitive control structures; non-determinacy in the order of evaluating guards within a guarded command structure; explicit scope rules with which to implement the author's ideas about "separation of concerns"; a syntactically distinguished initialization statement for all program variables; the total absence of a "go to" feature; and an array mechanism which implements the author's view of the array as a function (a mapping from a domain, consisting of subscripts, to a range, consisting of values). The language, taken as a whole, presents a spare, coherent set of structures necessary to present significant programming examples for discussion. In particular, Dijkstra's presentation of the language excludes input/output mechanisms, procedures (and recursion), and data types or structures beyond scalars and single-dimentsion arrays of integers, characters, or booleans.

The patterns of reasoning advocated by Dijkstra, and illustrated in his examples, include _enumeration_, for reasoning about alternative statements and statements in sequence; _induction_, for reasoning about repetitive statements and developing loop invariants; and _mathematical abstraction_, for reasoning about programs at various levels of their stepwise refinement.

It may be impossible to quantify the influence of Dijkstra's writings on programming methodologists. However, there can be no mistaking the impact of _A Discipline of Programming_ on the computer science community.

In 1977, a proposal was made to build a translator for Dijkstra's language and to evaluate the effectiveness of Dijkstra's approach in teaching introductory programming classes. In commenting on available programming languages for introductory instruction[136] and the potential benefit of implementing Dijkstra's language, the proposal stated

> The inadequacy of present programming methods and tools has nowhere been more clearly evidenced than in elementary and intermediate programming courses. The fundamental concepts of algorithm construction are obscured by the complex features of realistic programming tools. Simplified versions of languages impose arbitrary restrictions on the programmer. These restrictions also obscure the fundamental structure of algorithms. Much of the time that should be devoted to teaching the construction and evaluation of algorithms is spent instead on teaching how to get around a programming language and system.[137]

As a remedy to that state of affairs, a translator would be constructed for Dijkstra's programming language (hereafter to be referred to as DPL); suitable course materials would be developed for communicating Dijkstra's ideas on programming to introductory-level students; and the effectiveness of applying these ideas to introductory instruction would be evaluated in a controlled experiment.

---

[136] At the time of the grant application, PL/C was being taught in introductory programming classes as the University of North Carolina.

[137] D.L. Parnas, unpublished grant proposal, 1977.

The proposal was funded in September, 1977.[138] A translator for Dijkstra's language was developed by members of the department,[139] and became operational for testing in 1979. Trial runs on using the DPL approach in a classroom situation were conducted on a semester-long basis during the Spring, 1980 and Spring, 1981 semesters, each time with about 25 volunteers selected from the roster of the larger standard introductory programming course. Plans were then made to continue onward to a formal, controlled experimental evaluation of the approach, to be conducted during the fall of 1981. The description of that experiment follows.

3.2 GOALS OF THE PROPOSED RESEARCH

The goal of the dissertation research discussed here was the objective evaluation of an educational approach to teaching computer programming which emphasizes development of assertion-based arguments of program correctness hand-in-hand with the development of the programs themselves, and utilizes a language which supports that approach. That evaluation was conducted by comparing the effectiveness of the new approach to that of a conventional time-tested approach which emphasizes program testing and utilizes Pascal, a widely-distributed, general-purpose language commonly regarded since the mid-1970's as the best pedagogical language.[140]

---

[138] National Science Foundation grant number SED77-18518.

[139] John Bishop, "The Portable DPL Compiler Project," Master's thesis, Technical Report TR80-008, Department of Computer Science, University of North Carolina, Chapel Hill, 1980.

Karl Freund, "The Design and Abstract Specification of a Translator Module," Master's thesis, Technical Report TR79-012, Department of Computer Science, University of North Carolina, Chapel Hill, 1979.

James George, "An Abstract Machine as an Aid to Compiler Portability," Master's thesis, Technical Report TR79-017, Department of Computer Science, University of North Carolina, Chapel Hill, 1979.

Dan Lambeth, "Use of Trace Specifications in the DPL Compiler," Master's thesis, Technical Report TR79-019, Department of Computer Science, University of North Carolina, Chapel Hill, 1979.

[140] K. Jensen and N. Wirth, Pascal User Manual and Report,

The primary differences between the DPL approach and the conventional approach, using Pascal, were seen as the following:

1. Both approaches teach solution techniques and algorithm development. However, the conventional approach uses program testing and hand simulation of program execution as the primary means of verification, and relies on a language implementation whose compilation produces diagnostics for all syntax errors contained in a program and whose execution produces partial output even for many incorrect programs. The DPL approach uses informal correctness arguments as the primary means of verification, and embodies a philosophy that program compilation should report only the first syntax error encountered and that execution should produce output only for correctly terminating programs.

2. The conventional approach employs, for examples and problems, a general-purpose programming language which contains data structuring capabilities, control flow mechanisms, and input/output facilities necessarily sophisticated enough to satisfy its general-purpose user community. DPL was designed for teaching and expository purposes, contains only a small set of language mechanisms, and provides only the most primitive of input/output and data structuring facilities.

3. The conventional approach relies on the implicit semantics of the selected language and the understanding of its proper usage which the student picks up from looking at textbook examples and from experience. The DPL approach provides explicit semantics for the DPL language.

Thus, plans were made during late spring and early summer of 1981 to conduct a carefully controlled teaching experiment during the following fall semester. At the same time, however, administrative decisions by the university computation center concerning machine support for introductory instruction afforded an opportunity to include an additional aspect in the planned experiment. Sufficient funds were allocated[141] to acquire a number of Apple[142]

(New York: Springer-Verlag, 1975).

[141] University funds were matched to support provided by the

microcomputers for the purpose of switching introductory programming instruction from the large centralized campus and Triangle Universities Computation Center (TUCC) computing facilities. Conversion of introductory programming courses to microcomputers had long been advocated locally by some who felt that the quicker turnaround time and more personal nature of computer interaction would benefit beginning students. Convincing arguments were made that the switchover to microcomputer-based instruction would be better accomplished in stages over several semesters, initially with a modest number of students, and furthermore, that comparisons between students using the micro-based UCSD Pascal system and students using a Pascal compiler on a large batch machine might provide interesting results.

Therefore, the resultant experiment emerged as a dual experiment, with the conventional approach/batch Pascal group serving as control for two experimental groups: the DPL group, using a contrasting program development methodology and (batch-processed) programming language, and the Apple group, using a conventional program development methodology and nearly identical language but a contrasting mode of machine access. Details of the experimental design, computing environments, and experimental procedures will be provided in later sections.

National Science Foundation instructional equipment grant #1-0-110-3276-XA584.

[142] "Apple" is a trademark of Apple Computer Co. For the sake of brevity, all future references in this dissertation to the students using the UCSD Pascal system running on the Apple microcomputers will be to the "Apple section". While similar educational outcomes might be achievable with other manufacturer's microcomputers, no other vendor's microcomputers were used in this experiment. Since the observed results might not be generalizable beyond the specific system utilized, namely an Apple II computer running UCSD Pascal, it seems appropriate to identify that subject group with the specific system used. No endorsement of the company's computers is intended beyond that implied by the statistics reported herein. No financial support for this experiment was supplied by Apple Computer or by the suppliers of the Pascal system software, nor was any direct communication conducted with the vendors. The systems used were obtained under existing state purchasing contracts.

3.3 HYPOTHESES

The following hypotheses were formulated with respect to the group using the experimental DPL approach and the conventional batch Pascal group:

1. That on programming assignments, students in the DPL group would submit fewer runs having unintended results than would students in the conventional group on the same problems;

2. That the DPL group would require less debugging time (time expended after the first machine run until completion of the assignment) than would the conventional group;

3. That when each student had complete the assignment to his satisfaction, a higher percentage of DPL students' programs would actually run correctly, according to problem specifications, on independently supplied test data, than would programs from the conventional group;

4. That the programs of the DPL group would be of greater simplicity, according to the McCabe complexity metric, than would the programs of the other group;

5. That the DPL group could learn to program in the language the other group studied, in a condensed time period at the end of the semester, and that on the last problems assigned to both groups, a higher percentage of the DPL group's Pascal programs would be correct, according to problem specifications, than would the programs of the conventional group.

Similar hypotheses, with the exception of the fifth one, were formulated for the experimental Apple section in relation to the control group, the batch Pascal section. In particular, it was hypothesized that the Apple section would require less debugging time than the conventional group and that a higher percentage of its students would produce correct programs.

Students were instructed that their primary objectives in solving assigned problems were to develop correct solutions that were well-documented and as clear and readable as possible. No objectives were specified in regard to machine resource usage or time expenditure on the students' part.

3.4 EXPERIMENTAL DESIGN

The experimental design was a two-way, mixed, between/within subjects design. The class section (DPL, batch Pascal, Apple Pascal) was the between-subjects variable and the programming problem assignment was the within-subjects variable. The individual student was the basic unit of analysis. Dependent variables were the measures such as program correctness percentage, number of runs with unintended results, and time expended for each student on each problem.

Different students, therefore, were assigned to different sections. The "repeated measures" aspect of the design comes from the fact that measures of performance were obtained for each student within a given section on each problem. Repeated measures designs are commonly used in educational experiments where the effect of prior experience on future learning is of primary importance.[143]

Students were randomly assigned to class sections after the first course meeting. (Since there was only one course section of introductory programming in which students could register, all introductory students went into the same "pool" for subsequent assignment; there was no chance of subject self-selection into sections on the basis of class schedules, which might be biased by academic major, age, employment, etc.) Approximately-equal thirds of subjects were assigned to the DPL section, the batch Pascal section, and the Apple Pascal section. The DPL section was taught by this author, and the two Pascal sections were jointly taught, in a common lecture section, by a member of the computer science department faculty.[144] Enough students were assigned to each of the three sections, about 85 apiece, that although one lecture hall had 85 students and the other 170, both could be regarded as large lecture sections.

[143] See G. Keppel and W.H. Saufly, Jr., Introduction to Design and Analysis, A Student's Handbook, (San Francisco: W.H. Freeman and Company, 1980), Chapter 8, for material on repeated measures designs.

[144] The Pascal sections were taught by Dr. Stephen M. Pizer, Professor of Computer Science and Director of Undergraduate Studies for the Department of Computer Science. Philip Koltun, who taught the DPL section, had a teaching background that included five years' full-time faculty experience at the university level, the last three as Assistant Professor of Computer Science, as well as several years' work as a teaching assistant. Both Dr. Pizer and Mr. Koltun had taught programming courses before. In particular, Dr. Pizer had taught programming using Pascal before, and Mr. Koltun had taught programming using both DPL and Pascal before.

Class size, then, was not regarded as an additional independent variable.

The design question of whether to have the same instructor teach both lecture sections or whether to have different people do so was confronted at an early stage. While the final decision might be criticized for introducing the confounding factor of instructor's influence, the alternative might be equally suspect: A single instructor's methodological biases might influence his presentation and the differences in the two methodologies might prove difficult to maintain sharply in focus. Ultimately, the decision was reached that each lecture should be delivered by someone who firmly believed in the methodology being used. No determination of the actual instructor's effect was quantifiable.[145] The ideal experimental design, with a statistically suitable number of instructors randomly assigned to class sections, was clearly impractical. The only recourse, for the reader, is to keep the experimental limitations in mind when considering potential generalizations of the research results.[146]

In summary, the commonly-shared attributes of each of the experiment's sections are presented below:

```
DEL             Batch Pascal       Apple Pascal
 |                  | |                  |
 ----------------     ----------------
         |                   |
         |                   |
Batch Processing       Language
Problem Assignments    Instructor/Lectures
                       Program development
                           methodology
                       Problem Assignments
```

Figure 1: Shared attributes of experimental and control sections

[145] Questions on a mid-semester questionnaire shed some light on the effect of instructors and course assistants. See Appendix 8.1.

[146] Many such compromises must be made when doing experiments involving actual classroom situations. Strong restrictions are imposed by having student subjects registered for course credit and by having limited financial and instructional resources with which to work. Reflections on such experimental considerations will be made in Chapter 7.

3.5 EXPERIMENTAL PROCEDURE

3.5.1 Assignment of Subjects

As noted previously, only one section of introductory programming was available for student registration. During the first class meeting, a sign-up sheet was circulated, asking for student preferences as to which of six different Wednesday or Thursday afternoon or evening lab session times they preferred. Working from an alphabetized version of that sign-up sheet roster, students were randomly assigned to each of the Apple Pascal, batch Pascal, and DPL lecture sections. After that initial assignment, students were distributed into lab sections within each methodology (lecture section) according to their time preferences and the need to balance each teaching assistant's student load.

Biographical questionnaires were distributed at the second class meeting, the first time the students met in separate lecture sections. A summary of subject characteristics by section is presented in Table 1.

TABLE 1

Subject Characteristics by Section

(Second class meeting)

	Apple	Batch Pascal	DPL
NUMBER OF SUBJECTS	91	97	86
SEX			
Male	54%	57%	48%
Female	46%	43%	52%
MAJOR			
Computer Science	26%	20%	19%
Mathematics	22%	12%	16%
Other	52%	68%	65%
YEAR			
Freshman	17%	10%	9%
Sophomore	30%	33%	36%
Junior	30%	23%	26%
Senior	18%	26%	20%
Graduate student	6%	6%	2%
Evening college	0%	2%	7%
GPA	2.96	2.95	2.94
SAT VERBAL	542	534	533
SAT MATH	604	577	586
* Good experience with			
Calculus 1	79%	74%	79%
Calculus 2	40%	38%	45%
Logic	11%	15%	13%
Writing	90%	89%	88%

* "Good experience with" means that the subject completed the course in college or high school with a grade of "C" or better.

3.5.2 Course teaching assistants

Two master's students from the Department of Computer Science were assigned as teaching assistants to each of the three major experimental or control sections. Each T.A. was responsible for two laboratory sections, resulting in a ratio of approximately 46 students to each assistant at the outset of the semester. Assignment of assistants to the three major sections was made by the department chairman; each major section received one experienced and one first-time teaching assistant.

In addition to the teaching assistants, whose time commitments of twenty hours per week included four hours per week of open consultations with students in the computation center, there were also four undergraduate assistants hired to provide an average of fifteen hours per week of open consultation in the computation center. In addition, seven other computer science graduate students provided one hour per week of open consultation apiece.

3.5.3 Computer access

The primary facilities used by all introductory programming students are located in the basement of Phillips Hall on campus. Several remote job entry stations scattered around the campus were also available to students. In the Phillips Hall facility, keypunches used by virtually all students in the batch Pascal and DPL sections[147] were located within 75 feet of the dispatch window to which card decks were submitted and from which output and card decks were retrieved. Turnaround time for such access usually varied from ten to thirty minutes.

Another fifty feet down the hallway was located the room containing all nine Apples plus a table for the student teaching assistant holding open consultation hours. That room was staffed roughly from 10 a.m. to 11 p.m. weekdays and somewhat more-restricted hours weekends, so that help was immediately available, on a first-come first-served basis for any Apple or batch processing student who needed

---

[147] While terminal-based editing and remote batch submission of introductory programming jobs was possible, no mention of such possibility was volunteered by the instructors nor was any instruction given, upon request, in the usage of such system. It is estimated that, at most, a handful of students from the batch processing sections might have submitted their jobs in this manner.

system usage or programming assistance. The Apple computers, themselves, were accessible on a 24-hour basis, as was the batch processing system. A mechanism existed to permit Apple students to reserve up to four half-hour sessions within any consecutive three-day period, space permitting. Students could work at an Apple at other, unreserved times provided that no one else had reserved that machine.[148] An interface to the batch subsystem permitted Apple students to route files containing source listings and execution output to the main computation center printer for hard copies of their programs; attempts at keeping an inexpensive printer running in the Apple room proved largely unsuccessful.

The Apple system itself consisted of a 64K Apple II running UCSD Pascal, with dual disk drives at each station,[149] and a 40-column monochrome display. Compilation and subsequent program execution, including input data entry, took roughly three to four minutes on average. Instruction on how to use the Apples was conducted by the graduate teaching assistants in the regularly scheduled lab sessions for the Apple section.

3.5.4 Course materials

The first day of class, students were given a general information handout describing course objectives, content, and requirements, and a letter informing them that they would be participating in a formal evaluation of several different methods of teaching introductory computer programming. The letter told them, in general terms, that an educational experiment was being conducted, but shielded them from details of the experimental hypotheses and identities of the experimental and control groups. Students were told that their course grades would be unaffected by any of the information they would be asked to provide concerning the effort they expended in solving problems and the intermediate outcomes of their work. Furthermore, they would be evaluated only in relation to other students in their particular section, so that preoccupation with the section to which they were assigned could be minimized.

A biographical questionnaire eliciting relevant background information was administered the second day of class. Anonymous mid-semester and end-of-semester

[148] Frequency and duration of Apple sessions are reported in Appendix 8.2.

[149] The following semester a Corvus hard disk was installed to store system files, thus reducing the individual requirements to one drive per station.

questionnaires were also given out. Consult the appendix for forms and tabulated results.

Students in all three sections were required to purchase the Conway, Gries, and Zimmerman text,[150] _A Primer on Pascal_. Additionally, the Apple section was required to purchase a second text[151] for UCSD Pascal information. The DPL students were asked to buy the Conway, Gries, and Zimmerman primer for use at the end of class, and were given an extensive set of class lecture notes and a DPL manual for use during the first portion of the course.

3.5.5 Lectures

All sections received lecture presentations on computing systems, algorithm development, and algorithm expression in a particular language. Programs presented in the DPL section were developed in stepwise fashion using logical assertions to reason about what needed to be accomplished. Informal arguments were given to verify the correctness of the developing program at each level of refinement, and postconditions were used to determine what the program actually accomplished. Programs presented in the Pascal sections were developed in stepwise fashion using narrative prose and hand simulation of execution to verify the correctness of the developing program. Examination of simulated output was used to determine what the program accomplished.

Lecture classes met twice a week for about one hour each. A summary of the lectures presented is given in Appendix 8.3; the individual hourly examinations which were intended to reinforce the approach to the lecture material are given in Appendix 8.6.

The switchover from DPL to Pascal for the DPL section was accomplished at the two-thirds point of the semester. Similarities and differences between DPL and Pascal, and programming concepts embodied in the more general-purpose language, were presented in lecture material to the DPL section. Discussion of detailed Pascal syntax and input/output peculiarities was conducted in the laboratory sessions. Sample programs and algorithm development in Pascal were presented, as before, using correctness arguments.

---

[150] R. Conway, D. Gries, and E.C. Zimmerman, _A Primer on Pascal_, (Cambridge: Winthrop Publishers, 2nd Ed., 1981).

[151] K. Bowles, _Beginner's Guide for the UCSD Pascal System_, (New York: Byte/McGraw Hill, 1979).

3.5.6 Programming assignments

A common set of programming assignments was negotiated between the two instructors so that neither the DPL section nor the Pascal sections would have an advantage ascribable to language structure or programming style. No attempt was made to take advantage of any of the special characteristics of the Apple microcomputer, such as graphics processing or sound generation.

After an initial assignment of copying and running a handout program, five programs were subsequently assigned with either a one-week or two-week solution period, each section writing the programs in its designated language. All of these five problems, except the last, introduced some new language structure or additional level of control flow complexity.

After the last assignment of this phase of the course, the DPL group accomplished the switchover to Pascal with two practice Pascal programs during a two-week period. One of these programs involved re-coding into Pascal a DPL program presented in the lecture notes. The other involved novel features of Pascal arrays. At the same time, the Pascal groups were working on a single programming assignment involving pattern matching.

Finally, two more programming assignments were presented in identical form to all sections for solution in Pascal during the last three weeks of the semester. A summary of the assignments is presented in Appendix 8.7.

Problem descriptions were handed out in lab sessions. Completed assignments were due back in either a one week plus one day time period, or two weeks plus one day, thus permitting at least one more lab meeting before the end of the solution period. All students were required to turn in a run analysis sheet[152] with every assignment, on which they were to keep track of

1. the number of runs used;

2. the outcome of each run (including whether the result was an intended or unintended one and the nature of errors, if any);

3. the objective of that run, whether to test a complete solution to the problem or a partial solution, or to discover how some language feature worked;

[152] See Appendix 8.4 for a blank form.

4. a prose description of the changes made in the program since the previous run; and

5. the number of hours expended before and after the first run.

In addition, to reinforce the particular methodologies being used, the DPL students were required to turn in the informal correctness arguments they had developed along with their programs[153] and the Pascal students were required to turn in the test data on which they expected their programs to produce correct results.

When the students were satisfied that their programs were correct they were asked to submit to the teaching assistants their programs, in the form of card decks for the batch processing sections and diskettes (to be copied onto a master diskette) containing object code for the Apple section. Subsequently, the submitted programs were run on test data the instructors jointly prepared to exercise as many aspects of the student programs as possible. In no case was the test data released before the assignment due date. The teaching assistants were given the resultant source code and output listings for grading of stylistic content and correctness, and at that time were asked to record the correctness percentage and McCabe complexity metric[154] required for evaluation of the experimental hypotheses. Graded programs were returned to the students. However, the students were never told anything about the recorded McCabe metric values. Similar subjective grading criteria were used across the three major sections, with small variations in weightings according to the biases of the two instructors.

An unfortunately high number of bugs was found in the DPL compiler during the course of the problem solution periods. Among those bugs were incorrect handling of program scope units, of nesting of guarded commands, and of array domain operators, and production of misleading error messages.[155] Approximately half the students in the DPL

[153] See Appendix 8.5 for an example of an instructor-developed correctness argument for a simple algorithm.

[154] The McCabe metric reflects the control flow complexity or complexity of decision structure in the program. The metric may be quickly calculated as one more than the number of conditions in the program. See the earlier section in the literature survey, on measures of program complexity, for more general information about the McCabe metric, and see Appendix 8.8 for instructions on computing its value.

section had some encounter with compiler bugs during the semester, while virtually none of the Pascal students did, a scattering of complaints concerning the Apple session-scheduler being the only difficulty encountered in those quarters. Thus, an unintended source of confounding was introduced into the experimental design: relative goodness of the compilers being used by the DPL and Pascal sections.

155 A complete list of detected compiler bugs is given in Appendix 8.9.

Chapter IV

DATA COLLECTION AND REDUCTION

4.1 DATA COLLECTION PHILOSOPHY

A decision, based on both philosophical and pragmatic grounds, was made to ask students to report primary data themselves, on their programming efforts, rather than automatically capturing that data without their knowledge. It was the experimenter's strong feeling that a student's privacy in computer usage should not be involuntarily compromised, any more so than should the student's privacy in selecting certain materials for study in a university library be compromised. In particular a student's privacy should not be violated just because the nature of the computer makes it possible to carry out surveillance without detection. Furthermore, it was felt that informing students what was needed from them and how it related to the experimental objective of improving programming education would help enlist their cooperation in providing accurate and candid information. This was viewed as particularly important for data and subjective opinions that could be collected only by directly requesting it of the subjects.[156]

On a pragmatic basis, because both batch and microcomputer systems were used, different data collection mechanisms would have been needed, which likely would not have been equally unobtrusive. (All batch jobs submitted for execution could easily have been "drained" to an archival tape for later analysis; however, given the microcomputer configuration used, intermediate versions of the Apple programs could not have been stored away without significant degradation of response time.) Therefore, the decision was made to ask students to record measures of their programming effort on standard run analysis sheets. Furthermore, it was felt to be educationally advantageous (though not directly quantifiable) for students to be aware of their own programming behavior.

---

[156] The research proposal, including experimental design and descriptions of data to be collected, was submitted for prior review and approved by the Graduate School's office of research, which had access to the university's official Human Subjects Committee.

The decision to collect card decks for the final grading run and manually insert data cards for the batch groups was motivated by the difficulty of simultaneously preventing premature unintended access to the final instructor-designed test data, permitting late students to work on programs past the deadline,[157] and getting graded programs back to students as promptly as possible. An undergraduate student was employed twenty hours per week solely to assist in collecting and running and returning the card decks, diskettes and listings.

4.2 RUN ANALYSIS SHEETS

Preliminary versions of the run analysis sheet were tested in earlier course offerings with both DPL and Pascal students. A narrative description of changes made from run to run was used in an early version to determine the categories of execution outcomes for inclusion on a later version. That feature was retained on the back of the present version[158] in order to verify and disambiguate the responses on the front, and was indispensable for that purpose as well as for permitting students to communicate their frustrations or exultations. As it turned out, in filling out the forms many students failed to distinguish language from logic errors or aspects of the problem from aspects of the solution, so the narrative comments were quite helpful. The subjective impression of the experimenter was that those failings were less prevalent in the DPL section than in the Pascal sections, a possible outcome of the experimental emphasis on logical reasoning about programs.

Many difficulties arose, on the part of the experimenter examining the run sheets, in trying to classify errors as either language or logic errors, or determining if the intention of the run had been met. The experimenter, himself, checked the coding of answers and keyed the data from all the forms, so at the least, a high degree of consistency exists in the interpretation of student responses. Specific questions of interpretation are addressed below.

In regard to the reason for making a run, "testing a partial solution to a problem" was taken to be the motive only when it was evident that a separate program had been written to solve that subproblem or that scaffolding had been written to simulate the remainder of the whole program

[157] A penalty of about 12% a day was assessed for late programs, up to a maximum of three days late.

[158] See Appendix 8.4 for the run analysis form.

in order to test the subproblem solution in context. In particular, when only one bug evidently remained in the whole program, attempts aimed at ridding the program of that sole bug were still taken to be tests of the complete solution.

Interpreting whether the run did what the student intended could be aided by examining the indicated reason for making the run. A program was taken to do what was intended if it produced the correct output for a given input, even though it might produce incorrect output for some input on a later run. So a run (in Pascal) designed to produce debugging or trace output was taken as having an unintended result if the program printed incorrect output for the given input, even though the desired trace output might indeed also have been printed. In the same way, if a program run produced correct output for n-1 of its inputs but incorrect output for the nth input, the entire run was taken to have an unintended result.

The following remarks apply to the categorization of errors as being caused by either improper logic, improper language usage, or misunderstanding of problem specifications. Again, this categorization was often subjective, but at least applied consistently. In general, the information provided on the run sheets only permitted an error to be taken for what it seemed to be at run i, not what it turned out, in retrospect, to be at run i+j.

1. An "uninitialized variable" was a logic error, even though the occurrence might be detected (at least in DPI) during syntax checking.

2. A variable of the wrong type or an array with improper dimensions was evidence of a logic error. Using DIV (integer division) where / (real division) was needed, or vice versa, was treated as a logic error.

3. Assigning a new value to a declared constant was treated as a logic error.

4. Output appearance, if incorrect (not merely unattractive) in the student programmer's view, was treated as a logic error, provided that it appeared reasonable to expect the student had mastered the mechanics of the basic output statements. If that expectation was not reasonable, the deficiency was treated as a language usage error. (Failure to calculate the precise column in which a value would be printed was more often due to negligence or laziness than to ignorance or misunderstanding.)

5. Similarly, a lack of agreement in number or type of procedure parameters was treated as a logic error, provided that it appeared reasonable to expect the student understood the workings of the parameter passing mechanism, and as a language error otherwise.

6. Order of precedence errors could be interpreted as either logic or language errors, depending upon whether the narrative comments indicated the student misunderstood the underlying concept (language error) or understood the concept but misapplied the rules for expression formation (logic error).

7. Correspondence with begins and ends could also be interpreted either way, depending often on the stage of debugging at which the error occurred. Unmatched begins and ends were usually language usage errors and showed up early in the debugging; mismatched begins and ends were usually logic errors and showed up later in the debugging. However, substantial revisions in programs, though introduced to remedy logic flaws, often introduced new language usage errors.

8. Program changes for cosmetic purposes (for example, statement indentation) or documentary purposes (for example, header comments) usually maintained the previous run's intended result, though the additional run was often viewed as necessary to check that assumption. However, when errors were introduced, as a result, the errors were viewed as language usage errors.

9. Undeclared identifiers were treated as language usage errors.

10. Errors in job control language were treated as language usage errors.

11. Compiler bugs, which arose exclusively in DPL, were classified separately under a special code.

In general, even these coarse distinctions between language usage and logic errors were often difficult to apply. Frequently it was necessary to place oneself in the student's position as he might have formulated intermediate levels of refinement and ask whether the observed error was the result of bad design (logic error) or bad implementation (language usage error). As a result of classification difficulties such as these, any hypotheses that might be postulated in regard to specific features of language design and usage would surely be evaluated better in a specially designed experiment than in the context of a larger experiment such as this one.

The run sheet question that asked how much effort was involved in isolating the cause of an error produced little useful information. Typically another attempt at solution was made within an hour's time. It was difficult to determine, in general, at which run an earlier problem was finally resolved, and how much total effort went into that correction. Once again, a specially designed experiment would be better, in order to examine persistence and resolution of specific error types.

4.3 PROBLEMS IN DATA RECORDED BY THE TEACHING ASSISTANTS

Several of the teaching assistants failed to record the correctness percentage and/or McCabe metric data as they had been instructed to do. That failure was not detected until it was too late to remedy. In retrospect, final program executions on instructor-provided data should have been stored in archival form. However, because of the volume of programs involved (over 200 programs per assignment), that precaution was not taken as a matter of due course.

As a consequence, the correctness percentages for some of the Apple and some of the DPL students had to be estimated from the assigned program grade which also included (known) subjective criteria in addition to correctness. Because of this estimation procedure, the most reliable way to analyze the correctness data was to treat a student's program as being either entirely correct on the test cases or not entirely correct.

The difficulty concerning the McCabe metric involves both possible misinterpretation by the graders of the instructions[159] for computing the metric, and assumptions of questionable validity about the metric's computation, in the instructions themselves.[160] A further reason for treating

[159] See Appendix 8.8 for instructions on computing the McCabe metric.

[160] The instructions specified that the "Do-forever-with-exit-test" loop favored in some situations by the instructor of the Pascal groups should add two to complexity, one for the "While true do" part and one for

the reported McCabe metric values with caution is that the intuitive correlations between simplicity of program decision structure and program correctness that one would expect to see were not borne out in this experiment.

Because the programs were not stored away, as noted before, the metric cannot be recomputed at this point. However, a comparison between the metric values for the two Pascal groups should still be valid, as should intra-group examinations of the relationships between McCabe metric values and measures of programming effort, such as time expended in debugging.

4.4 POSSIBLE INACCURACIES IN DATA REPORTED BY STUDENTS

Since the students were relied upon for data concerning effort expended and outcomes of individual runs, a question arises concerning the accuracy of this information. Problems concerning categorization of errors in runs with clearly unintended results have already been addressed. A further awkwardness exists: With the later knowledge that a program was not actually correct (as evidenced by the output on the instructor's input test data) when the student thought it was correct, how does one now view the earlier report that a run's outcome matched the student's intention? The assumption was made in answering this question that the student was capable of discerning whether the output was correct for the specific inputs he supplied, and therefore, that his report of whether outcome matched intention should be taken at face value.

In general, however, interpretation of measures for programmers and programs which were not entirely correct proved much more difficult than data for correct programs: Was the given program incorrect because it lacked some critical code (resulting in a lower McCabe metric) or

---

the exit test. In retrospect, the "while true do" part should probably add nothing to complexity since, in terms of the metric's program flow graph interpretation, only one path may be taken after evaluating the "while" condition.

By similar reasoning about program flow graphs, DPL alternative statement constructions with two guards in which one guard is the negation of the other guard, should probably not, in retrospect, have both guards contributing to complexity, since the same number of control paths exit that construction as in the Pascal "if condition then statement else statement" construction, which contributes only one to complexity. (In DPL both guards must explicitly be stated.)

because it was too complicated (resulting in a higher McCabe metric)? Did the programmer fail to invest the requisite time or did he expend an inflated amount of time on ill-considered modifications?

A second area of concern about the accuracy of reported data focuses on whether the microcomputer section's students recalled details of their computer usage as faithfully as did the batch processing sections' students. A plausible assumption might be made that the Apple students' runs came in rapid succession, blurring the distinction between runs, while the batch students' runs were discrete events reinforced by an output listing after every run. However, counterarguments might be made on several grounds.

First, although the end-of-semester questionnaire[161] revealed that the Apple students were somewhat more likely to wait until the end of the problem period to record their data and had somewhat less faith in their own reporting of run data than did the batch students, they had faith in their own reporting of time data (owing, probably, to the discrete scheduling of Apple sessions) equivalent to that of the batch students. Furthermore, the responses on that same questionnaire indicate a consistent pattern of computer usage across all three sections. Students tended to make two trips a week to the computation center, spending either one to three hours there or more than three hours there, at a time. While there, students tended to make either three to five run attempts or six to ten run attempts. Thus, one could hardly support a view that the Apple students were firing off run attempts as fast as the machine would allow, thereby dimming their recall of individual run attempts.

A second counterargument to the concern that accuracy of recall by the Apple people was measurably different than accuracy of recall by the batch people might be based on results of relevant psychological studies, though the particular recall phenomenon of concern here does not seem to have been the subject of any studies. In experiments on the effects of repetition and exposure duration on memory, Hintzman[162] varied visual presentation of a series of words according to both frequency and duration, and reported that judgment of apparent frequency was highly correlated with actual frequency but relatively unaffected by duration. (Judgment of apparent duration was correlated with both frequency and duration.) This result might support a view that both microcomputer and batch processing students could be expected to report frequency of runs with equivalent

[161] See Appendix 8.2, questions 6-9, for details on the discussion below.

[162] Douglas L. Hintzman, "Effects of Repetition and Exposure Duration on Memory," Journal of Experimental Psychology, Volume 83, No. 3, 1970, pp. 435-444.

accuracy regardless of the turnaround time involved in a particular run.

A second study, by Madigan,[163] investigated word recall involving distributed repetition versus massed repetition of words. Although the study showed that recall is better if the repetition of an input is spaced further from the first presentation rather than closer to it, the differences in probability of recall even here, with a simple recall task and repetition lags measured in seconds not minutes, was at most fifteen to twenty percent. Thus, though generalizations from the cited psychological studies might be difficult to make, a case might be made that even if difference in recall of run frequencies and time expenditures existed between the microcomputer and batch processing sections, those differences would probably not be huge.

The final source of possible inaccuracy in data the students reported was the problematic performance of the DPL compiler.[164] Sixty-four percent of the DPL students reported some encounter with a compiler bug.[165] It is difficult to estimate how much of the total time expended on problem solution was devoted to trying to modify programs incorrectly translated by the compiler, the most serious of these situations being, of course, syntactically and semantically correct programs that were treated as incorrect by the compiler. Misleading compiler diagnostics can be a problem in any language.[166] But being unable to trust a specific error message when one is nevertheless certain that an error exists, seems qualitatively different than being unable to trust that a compiler has correctly translated one's program. More to the point, lack of confidence in the

---

[163] S.A. Madigan, "Intraserial Repetition and Coding Processes in Free Recall," _Journal of Verbal Learning and Verbal Behavior_, Volume 8, pp. 828-835, 1969.

[164] See Appendix 8.9 for listing of known compiler bugs.

[165] See end-of-semester questionnaire, question 11, in Appendix 8.2. The responses to the same question for the Pascal sections were also reported there, verbatim. However the reported encounter with a compiler bug by 48% of the Apple and 36% of the Batch Pascal students is viewed as unreliable, and attributable to either misleading error messages or problems in the Apple interface to the filer/editor or batch printer/scheduler subsystem.

[166] See, for instance, C. Litecky and G.B. Davis, "A Study of Errors, Error Proneness and Error Diagnosis in COBOL," who reported that 80% of a COBOL compiler's error diagnoses were misleading.

faithfulness of program translation undermines a beginner's trust that a methodology (the DPL methodology) employing formal reasoning to progress from specifications to implementation can result in successful programs. Thus the DPL section should be viewed as operating under something of a handicap due to compiler problems.

Chapter V

RESULTS

Before beginning this discussion, it would be worthwhile to reemphasize the basic nature of this study as a dual two-group experiment, with the batch Pascal section serving as a control group in each experiment. The comparisons between the DPL section and the batch Pascal section involved similar batch processing computer access modes and identical problem assignments, but different program development methodologies, different programming languages, and different instructors. The comparisons between the Apple Pascal section and the batch Pascal section involved identical program development methodologies, problem assignments, and instructor, very similar programming language dialects, and different computer access modes.

5.1 SUMMARY

The body of statistical run data and subjective student impressions supports the conclusions that

1. The DPL students significantly outperformed their batch Pascal counterparts through the end of the DPL part of their course, with respect to measures of program correctness and programming errors, but that the effect did not carry over to their Pascal programming experiences at the end of the semester.

2. The Apple Pascal students significantly and impressively outperformed their batch Pascal counterparts throughout the semester with respect to measures of program correctness, programming errors, time expenditure, and consistency of performance, and derived a higher degree of satisfaction from their learning experiences. Furthermore, the Apple mode of access had a noticeably beneficial effect on students of marginal ability.

5.2 DETAILS

5.2.1 Statistical Analyses and Data Transformations

Data analyses have intentionally been kept simple, rarely going beyond descriptive statistics, for several reasons, one statistical, one practical, and one relating to experimental design. The practical reason is that the large number of subjects utilized in this study gave us every chance for producing statistically significant results, no matter how small the differences in group means. However, statistical significance will not be enough to impress computer scientists, unless the observed differences are also methodologically and educationally important. Therefore, a guiding principle in presenting the analyses has been to lay out the group differences for the reader, advise when those differences were statistically significant, and caution when seemingly large differences were nonetheless lacking in statistical significance. The reader can then decide for himself what magnitude of group differences will impress him.

The experimental design reason for limiting the sophistocation of statistical techniques was alluded to in the earlier presentation of the design used here: Some people will object that the confounding introduced by instructor differences overrides all other concerns and that a true experiment of this nature should have a number of instructors randomly assigned to the different approaches. For those doubters, no amount of statistical wizardry will salvage a flawed design.

The statistical reason for limiting analyses to those presented is that in some instances, statistical assumptions about homogeneity of variances necessary for more sophisticated analyses of variance have been violated by the data, often along two dimensions (across problem assignments within a given methodological section, and across sections at a given problem assignment). As a consequence, the more sophisticated analyses would not, in some cases, be well-founded.

For the most part, in the presentations that follow, only two manipulations have been performed on the data. First, in order to reduce the number of data points that must be made sense of for each student, the seven common assignments from problem two through problem nine (with the exclusion of the dissimilar problem sevens) have been clustered into three subgroups of problems. In addition to permitting some smoothing out of the inherent variations in performance by averaging measures within each subgroup, the clustering makes sense pedagogically. Subgroup I (problems two through four) consists of introductory problems, subgroup II (problems five and six) consists of intermediate level problems and runs through to the end of the DPL part of the course, and subgroup III (problems eight and nine) are advanced problems for which all students wrote programs in Pascal.

Second, the students who finished the course have been partitioned into a subset called "consistent finishers" and a subset called "inconsistent finishers". The consistent finishers were the ones who got at least one problem entirely correct within each of the aforementioned problem subgroups. The inconsistent finishers failed to get at least one problem entirely correct in at least one of the aforementioned problem subgroups. The inconsistent finishers were regarded as qualitatively and quantitatively different in performance from the consistent finishers.

Qualitatively, the consistent finishers might be viewed as those people who received consistent reinforcement for applying the techniques of their particular methodology. (The reinforcement, here, was the gratification that comes with successful assignment completion.) The inconsistent finishers failed in some way to integrate all the lessons of their methodology at some point in the semester, with possibly adverse consequences for later learning.

Quantitatively, the number of problems solved entirely correctly by each subset differed markedly. In summary form, the number of problems (out of the original ungrouped seven) solved by each subset is given below in Table 2.

TABLE 2

Number of Problems Solved Entirely Correctly

	Consistent Finishers	Inconsistent Finishers
Apple	5.89	3.85
Batch Pascal	5.54	3.34
DPL	5.45	3.38
Minimum significant group difference	0.26	0.37

The "minimum significant group difference" referred to in Table 2 is a magnitude of three times the standard error of the mean for the entire collection of people in that subset. If the group means differ by at least this much, the means should be regarded as significantly different.[167] The number and percentage of students in each subset is also given below, in table 3.

TABLE 3

Size of Consistent and Inconsistent Finisher Subsets

	Consistent Finishers		Inconsistent Finishers	
	Number	Percent	Number	Percent
Apple	47	71%	19	29%
Batch Pascal	50	62%	31	38%
DPL	40	59%	28	41%

The final general observation to be made before presentation of results is that no extreme values were excluded from any of the subject-supplied data. Occasionally students reported 50 hours or 40 runs expended on a single one-week-long assignment. There was no a priori reason to exclude such measures, since characterization of difficult or even futile student efforts was of interest in this experiment, as well as average efforts. Furthermore, some of the students who reported such extreme allocation of resources evidently struggled through to successful completion of their assignments. They, too, should be allowed to make a contribution to the group averages.

---

167 The standard error of the mean in effect measures the within treatment variability. Therefore, if the group means vary by more than three times the standard error, the difference may be viewed as due to the treatment, not random variation. If the sampling distribution that the standard error represents were normally distributed, then a significance level of .05 in the usual two-tailed test would be equivalent to 1.96 times the standard error; a significance level of .01 would be equivalent to 2.56 times the standard error. So three times the standard error provides a conservative confidence level.

5.2.2 Correctness of Programs

As noted previously in table 3, a higher percentage of Apple students achieved consistently correct programs over the course of the semester than did the batch Pascal students (71% to 61%) . Meanwhile, the percentage of DPL students achieving consistent results was nearly identical to that of the batch Pascal section (59% to 61%). As table 2 showed, the average number of problems solved entirely correctly over the whole semester was significantly higher, for the Apple section, for both consistent <u>and</u> inconsistent subsets of students, than the average for the batch Pascal section. No significant differences existed, for either subset of finishers, between the batch Pascal and DPL sections, as measured over the entire semester.

But when the performance is examined through the end of problem six (the end of the DPL language part of the course for the experimental DPL section), a significant advantage for the DPL students over their batch Pascal counterparts can now be seen (Table 4):

TABLE 4

Average # of Correct Solutions Through Problem 6

	Entire Class	Number of Students
Apple	4.12	66
Batch Pascal	3.40	81
DPL	3.78	68
Minimum significant group difference	0.23	

An even greater advantage over the batch Pascal students can be found, however, for the Apple students.

The changeover from DPL to Pascal was not achieved as successfully as had been hoped possible. An examination of performance on problems eight and nine alone shows (Table 5) that the DPL students solved significantly fewer problems entirely correctly than did the batch Pascal students.[168]

TABLE 5

Average # of Correct Solutions for Problems 8 and 9

	Entire Class	Number of Students
Apple	1.15	66
Batch Pascal	1.28	81
DPL	0.78	68
Minimum significant group difference	0.15	

The following graphs display the fraction of entirely correct solutions within each group of problems, first for consistent finishers (Figure 2), then for inconsistent finishers (Figure 3). At each problem group point on the horizontal axis, vertical bars project the minimum difference that must exist between any two of the data points for that difference to be considered significant.

In brief summary, the problems in group one introduced the basic language structures, including alternative and repetitive statements. Problems in group two dealt mainly with algorithms requiring arrays, and problems in group three dealt with medium-length programs requiring subprogram modules and/or multi-dimension data structures. Consult Appendix E.7 for more details on individual assignments.

---

[168] Note, however, that the relatively low number of entirely correct solutions by the DPL section does not imply that they failed to learn Pascal adequately. "Entire correctness" is a very strict criterion; if an output label was misplaced on even one of the four graphs required as output on problem eight, for example, the whole program was counted as not entirely correct.

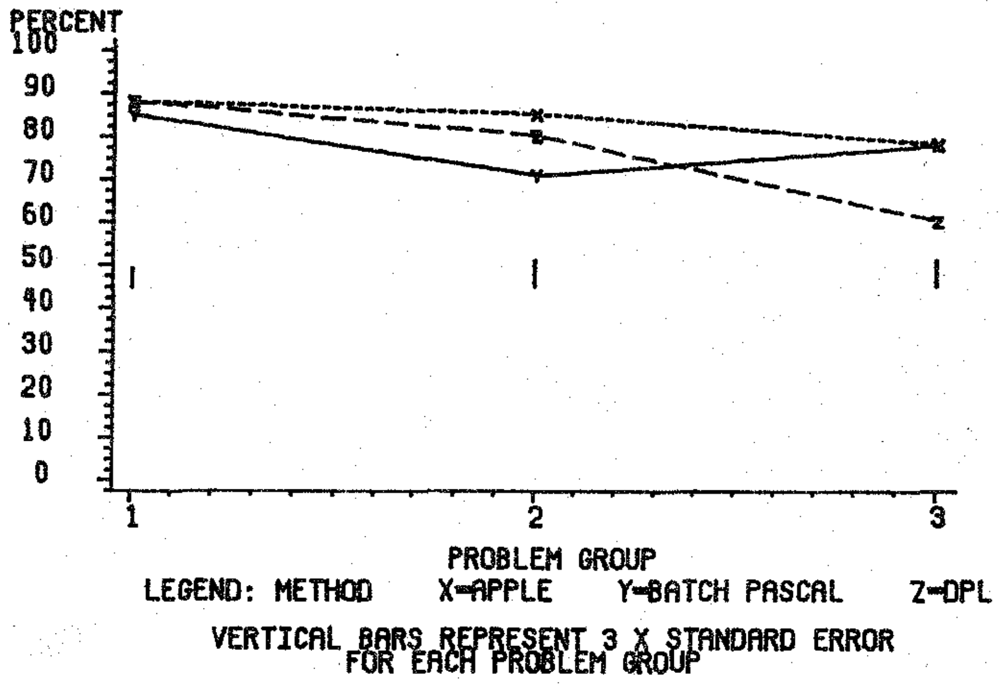

Figure 2: Average percentage entirely correct solutions for all programs by consistent finishers

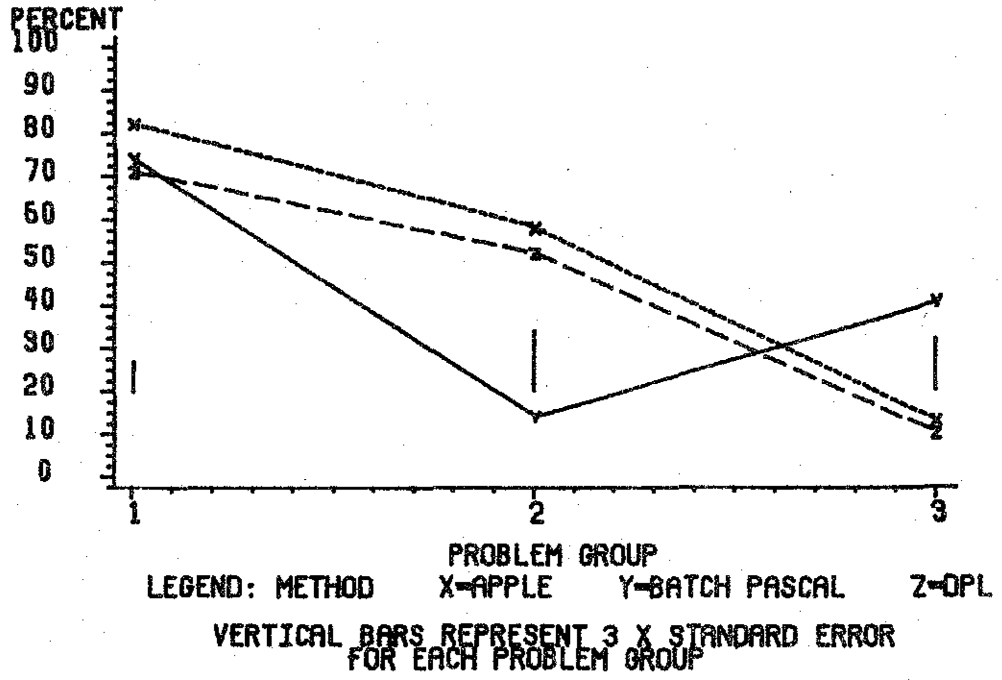

Figure 3: Average percentage entirely correct solutions for all programs by inconsistent finishers

5.2.3 Effort Expended by Students

Statistics presented in this subsection represent the machine and human time resources expended in solving assigned computer problems. The efficiency of those expenditures, in terms of the errors committed en route to a solution, will be presented in a later subsection. The graphs displayed at this point answer questions about the relative cost of each approach, and might be of interest to programming methodologists, computation center directors, and prospective students of programming concerned about the time required by introductory courses.

The statistics on average number of runs utilized include runs to test partial solutions, runs to discover how language features work, and runs at the end of the solution process to test presumably correct complete programs. Therefore, these statistics reflect efficiency of program testing strategies as much as assimilation of program logic and language rules. Consult statistics on runs with unintended results, presented in a later section, for a clearer reading of student understanding (or misunderstanding) of programming mechanisms.

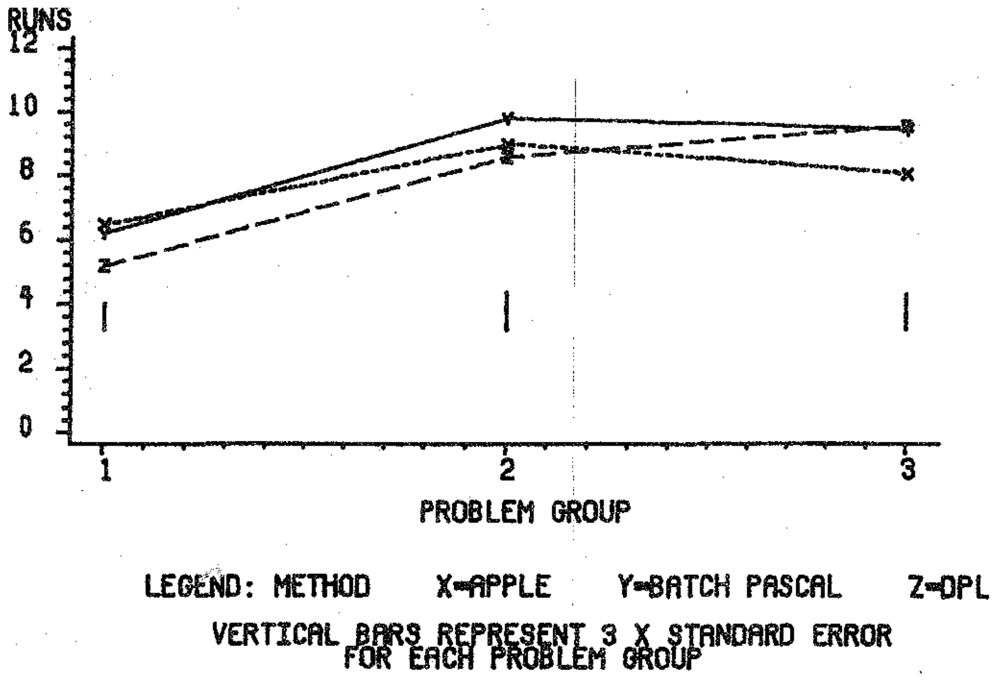

Figure 4: Average number of runs for all programs by consistent finishers

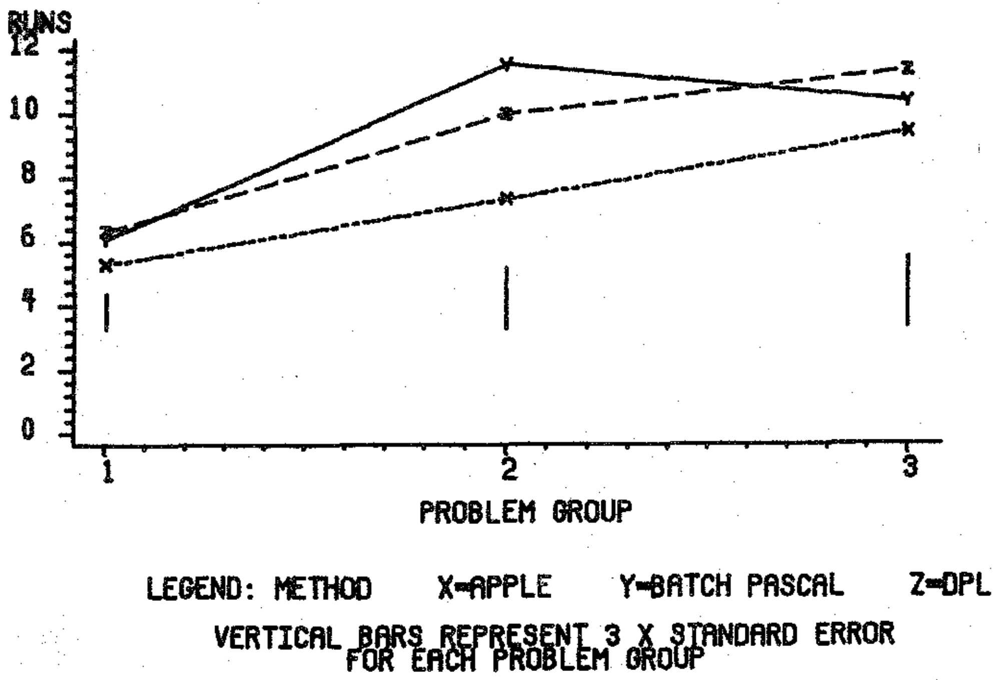

Figure 5: Average number of runs for all programs by inconsistent finishers

The graphs show that number of runs utilized generally increases with problem difficulty[169] and with the number of language mechanisms required in the program. Each assignment except the sixth one (which is represented in problem group two, in the graphs) introduced a new language feature. Figures 4 and 5 suggest that significant uncertainty about how particular language structures worked evidently continued well beyond the assignment that introduced each such structure.

Note that the number of runs utilized by Apple students generally improved (lessened in number), relative to batch Pascal students, as the semester went on.

The graph of hours expended preparatory to the first run (Figure 6) shows that very little difference exists between the three groups of consistent finishers.[170] The

---

[169] Consult question 15, end-of-semester questionnaire, in Appendix 8.2, for student estimations of problem difficulty, which in general agreed with the instructors' estimation of problem difficulty.

[170] No statistically significant differences existed at all, for the inconsistent subset. Where graphs are omitted, no significant or interesting group differences existed,

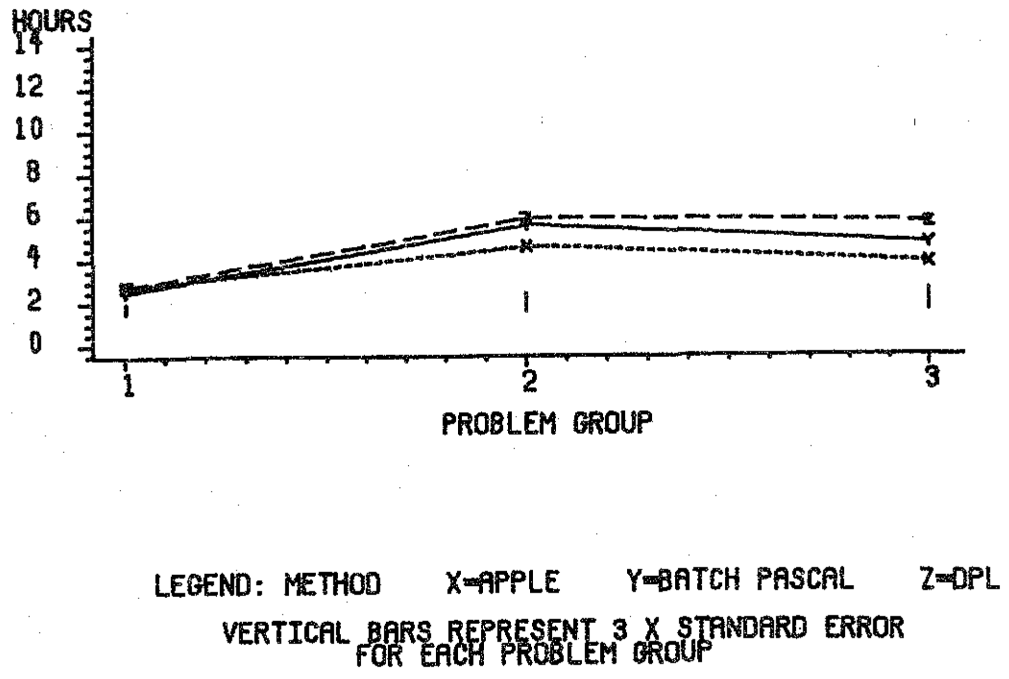

Figure 6: Average number of hours before the first run for all programs by consistent finishers

uniformity of the preparation time reported suggests a combination of possible influences: Difficulties inherent in understanding a problem and discovering a solution strategy dominate difficulties in expressing that solution in an algorithmic language, regardless of methodological differences; and student schedules permit just so much time to be allocated to solution design, regardless of instructors' preachings about trading off design time for debugging time.[171]

The most important showing of the "time before" graph, Figure 6, is that it disproves the reservation many educators share in regard to switching instruction to an interactive system: Students will not rush to the machine, counting on inventing programs on-line, without spending adequate time in designing solutions. Undoubtedly, the Apple scheduling mechanism, which limited students to

unless where noted otherwise in the text.

[171] Correlations between background variables such as grade point average and SAT scores and performance measures such as time before the first run suggest that the brightest students are also quickest. See the later section on characteristics of the student subjects.

reserving at most four half-hour sessions in any consecutive three-day period, contributed favorably to the observed behavior.

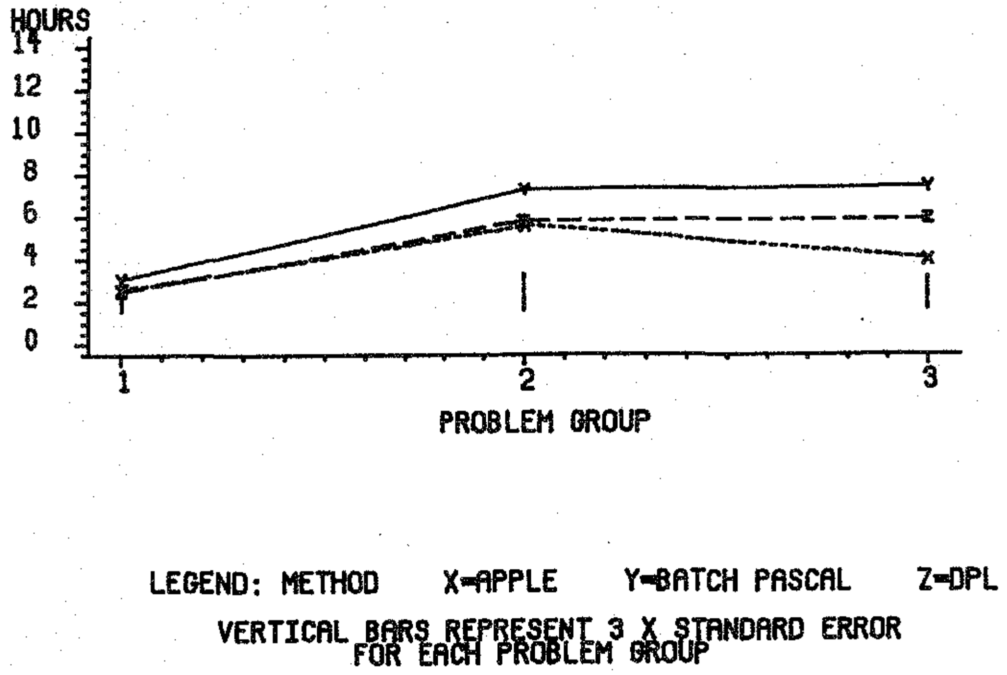

Figure 7: Average number of hours after the first run for all programs by inconsistent finishers

Number of hours spent after the first run measures the debugging time for the problem. DPL advantages over batch Pascal with respect to this measure, sit on the borderline of significance at every problem group. (See Figure 7.) However, the Apple group's advantage over batch Pascal not only increases to an impressive level but reflects a nice decrease in debugging time as the semester progresses.

The advantage of the Apple group can not merely be written off as faster turnaround time. Debugging requires thought as well as machine access. The degree of similarity between the batch Pascal and Apple sections in number of trips to the computation center, time spent once there, and run requests submitted per session, as reported on the end-of-semester questionnaire, suggests that the Apple people continued their computational sessions because they had the dual sense that they could wrap up the program right then and that their next effort at solution would be rewarded with immediate turnaround. Thus the nature of Apple usage produced a concentrated, intense effort, and it was that concentration, itself, which resulted in shortening the debugging period. No time was lost in recovering context every time the student came back to the program display after a lapse, as likely occurred in the batch processing sections.[172]

In summary, the total number of hours expended on each problem reflects no significant difference between the batch processing sections, but an advantage to the Apple section that increases as the semester goes on (when, presumably, the Apple student learns more about how to use and take advantage of the machine). See Figure 8.

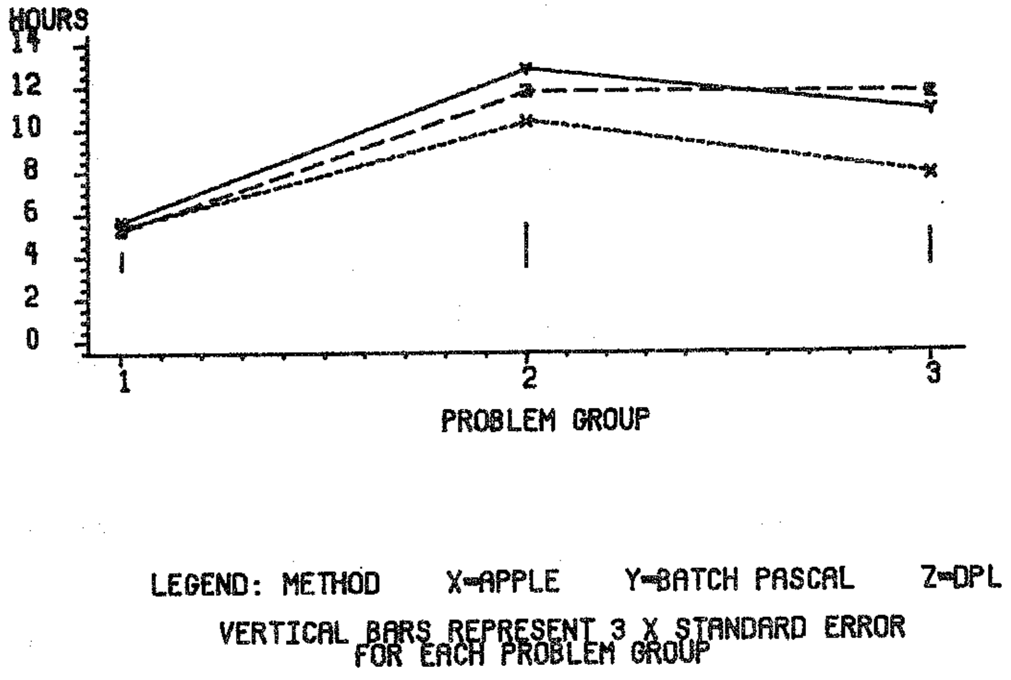

Figure 8: Average total number of hours expended for all programs by consistent finishers

While the total number of hours used by the batch processing sections may not have differed, the allocation of that time did. At each problem group reference point, the

[172] This observation would help distinguish the nature of microcomputer usage from that of time-shared interactive usage. The time-sharing system user is subject to the vagaries of system crashes, resource competition, and response time fluctuations, all of which the microcomputer user is shielded from.

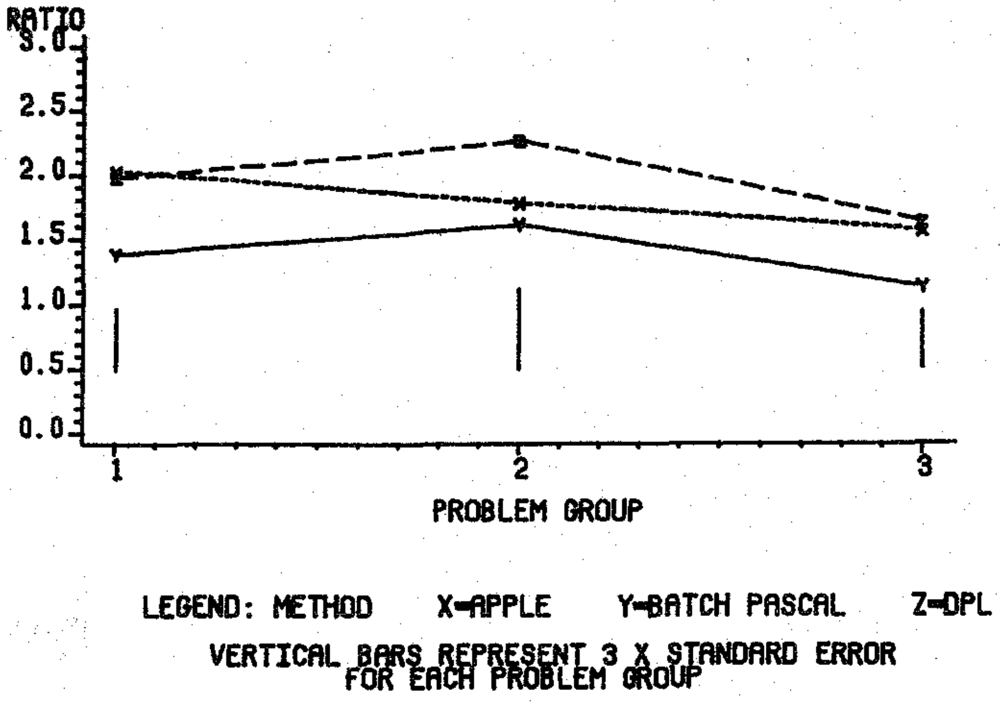

Figure 9: Average ratio of time before to time after first run for all programs by consistent finishers

DPL section had a higher proportion of time before the first run to time after the first run, than did the batch Pascal section. Subjective impressions during the semester suggest that DPL students with incorrect solutions spent too little time after the first run examining their results and thinking about their programs, or trying alternative test data.

Finally, it does not appear that any differences in comparisons of observed performance can be traced to the DPL policies of reporting only one syntax error and printing no partial output prior to abortive program termination. To the surprise of many, perhaps, the policies did not seem to place the DPL students at a disadvantage. (If there were such a disadvantage, one would expect a longer debugging period for the DPL people.) Neither did the policies seem to give the DPL section an edge. A possible conclusion is that the students in the other sections were not making use of all the error messages reported and/or were not producing trace output to help explicate program bugs. Students may try to track down only one error at a time. See, for example, Nagy and Pennebaker, "Automatic Analysis of Student Programming Errors," whose data led them to believe that "each new mistake is discovered only once a previous mistake has been corrected." Significant advantage might be yielded, therefore, in intensively instructing students in debugging techniques.

5.2.4 Errors Committed En Route to Solutions

The number of runs with unintended results reflects the student's understanding of how to develop correct programs and his assimilation of the details of programming language syntax and semantics. A run might have an unintended result due to a logic error (an error in the program's algorithm), a language usage error (an error in translating an algorithm into a language), a compiler bug (a system error in program translation), or a misunderstanding of problem specifications. To some extent, the number of runs with unintended results reflects how well the formal language descriptions and informal program examples can communicate the language's workings. To some extent, also, this measure reflects how well the particular programming methodology permits the student to uncover errors in his work along the way and to progress to an eventual solution. However, a word of caution is needed: Number of runs with unintended results used en route to the solution did not correlate significantly with eventual correctness of the finished product. So errors committed reflect efficiency of the solution process more than the quality of the final product.

This measure is unbiased by the goal of a particular run attempt, be it testing a complete solution, testing a partial solution, or discovering how a language feature works. Only runs with unintended results are counted. Consult section 3.2 for details of run result classification.

As Figures 10 and 11 show, below, consistent DPL students made significantly fewer runs with unintended results than did consistent batch Pascal students, but only through the end of problem group two.[173] That advantage did not continue once the DPL students switched to Pascal, and in fact both consistent and inconsistent DPL students made more errors in Pascal than they had in DPL. Possible conclusions from this pattern might include that DPL was less error-prone than Pascal or that the semantics of Pascal admitted of formal description less well than did those of DPL (language design issues); that Pascal was sufficiently different from DPL that the transference of programming language principles could not be easily effected (an issue of educational psychology); or that the details of Pascal were ineffectively presented (an issue of experimental presentation).

Apple students enjoyed a clear advantage with respect to their batch Pascal counterparts for this measure through at least the second problem group. The consistent Apple students continued their advantage through to the end of the

[173] This difference is even more impressive because the DPL statistic for problem group two includes about 0.5 runs with unintended results caused by compiler bugs.

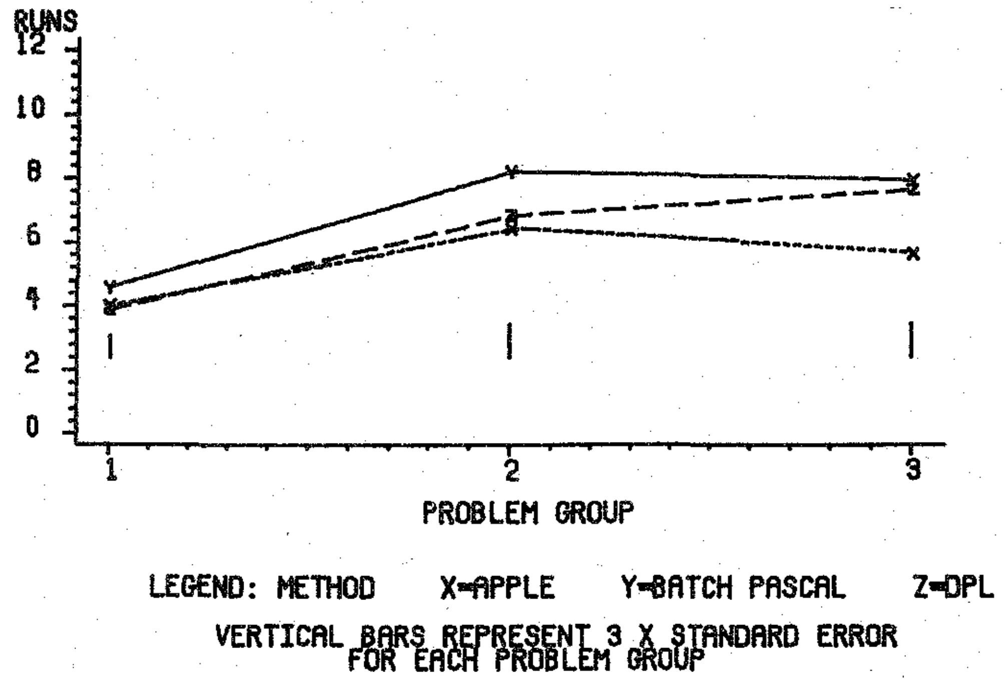

Figure 10: Average number of runs with unintended results for all programs by consistent finishers

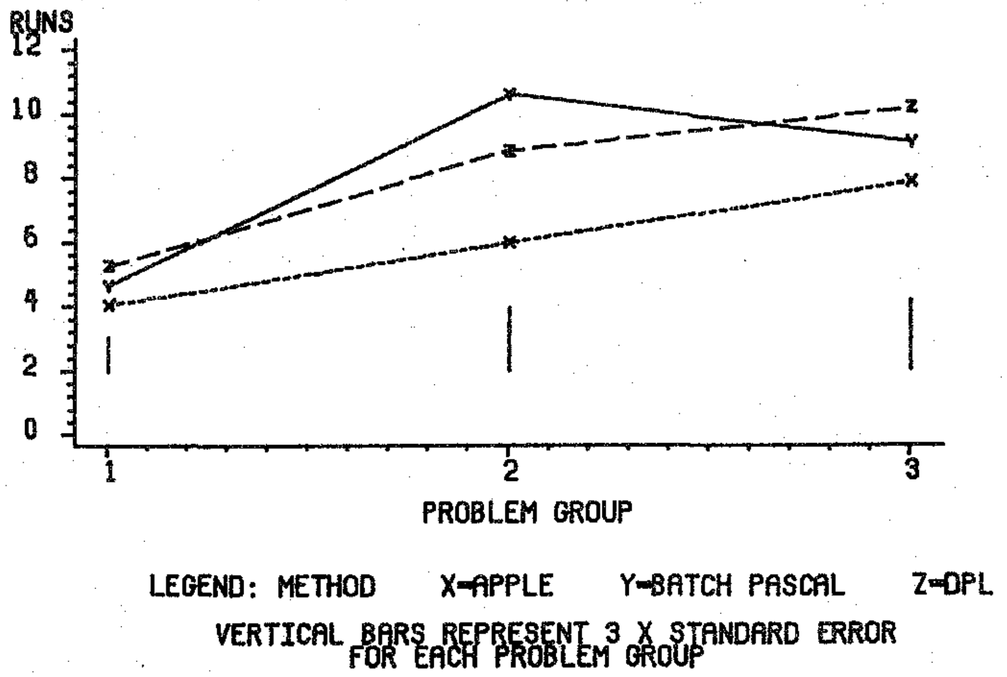

Figure 11: Average number of runs with unintended results for all programs by inconsistent finishers

semester, while students in the inconsistent Apple subset had substantially more difficulties in the third problem group than they had had previously. Since the third problem group required the longest programs, with some modularization necessary, it is possible that working without program listings most of the time[174] presented obstacles to some in the Apple group.

Note that students in the inconsistent subset generally had more runs with unintended results than students in the consistent subset did. However, the inconsistent Apple students evidently had more difficulties with the last problem group, as commented on above, while the inconsistent batch processing students (both DPL and Pascal) evidently had more difficulties with the problems of group two, which required array manipulation. By implication, the batch Pascal students in the inconsistent subset were in that subset because they had major difficulty with the second problem group; the Apple students in the inconsistent subset were in that subset because they had major difficulty with the third problem group.

With respect to particular types of errors that caused runs to have unintended results, the consistent DPL students for the most part outperformed consistent batch Pascal students with respect to both logic and language errors through the end of the second problem group. Consult Figures 12 and 13 below. However, those significant differences did not carry over to the third problem group. There were no significant differences between the inconsistent DPL and batch Pascal sections at any point on either language and logic errors, in part because of the inherently higher variance of statistics for these inconsistent subjects.

The Apple section came out best of all with respect to these comparisons. A very sizable advantage over the batch Pascal section was observed in regard to logic errors (owing perhaps to the greater concentration in effort extended by the Apple people, as suggested earlier), and a significant, if relatively small, advantage over the batch Pascal section in regard to language errors at an intermediate point in the

[174] No statistics were acquired during the semester on how often Apple students obtained listings of their programs. However, the following semester, in which all introductory students used the Apple systems, 57% of the students reported obtaining listing only every few sessions, 7% only once per assignment, 27% after every session, and only 9% once or more during an Apple session. Forty percent of the same students reported that the lack of a printout after every program execution caused them some difficulty. Failure to obtain a listing more often was attributable primarily to the slow turnaround time on printing.

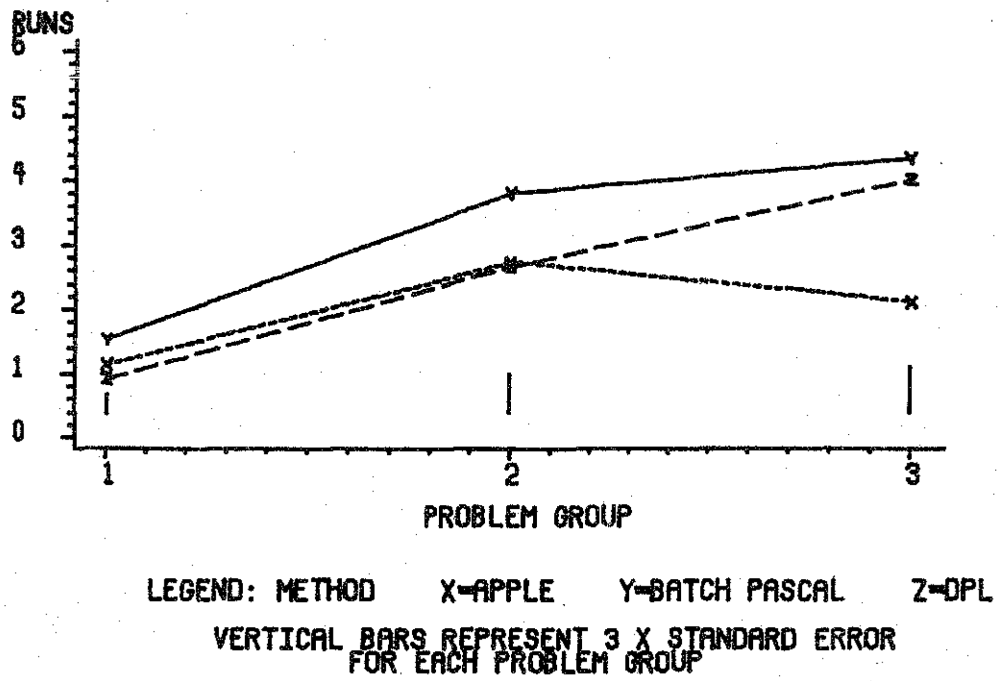

Figure 12: Average number of runs with logic errors for all programs by consistent finishers

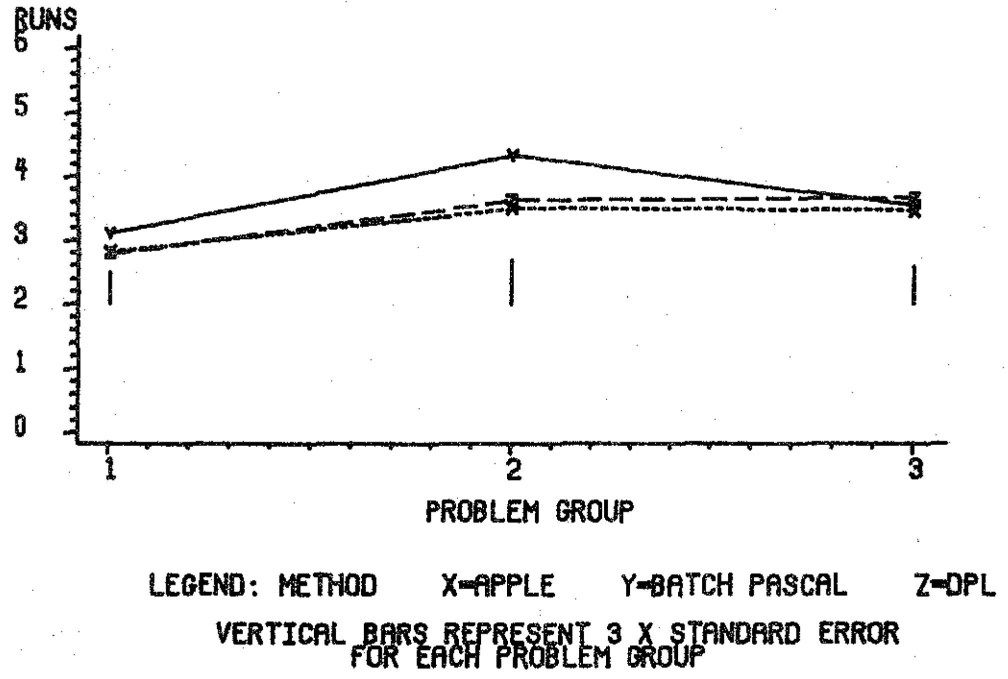

Figure 13: Average number of runs with language errors for all programs by consistent finishers

semester.

A most interesting set of responses arose from the end-of-semester question asking how many times a clinic attendant or teaching assistant had been unable to help with a problem in the student's program. Despite the fact that none of the clinic attendants had ever written a DPL program and that only two of the six graduate teaching assistants (the DPL assistants) had done so, the DPL students were unable to obtain help with their programs fewer times than students in either of the other sections.[175] The conclusion may be reached that DPL's language mechanisms have some intuitively understandable structure and/or that fewer severely contorted programs were produced using the DPL language and methodology than under the alternative approach. Either fewer problems were being brought to the attendants by the DPL students or else those problems which were inexplicable to the novice student could be easily unraveled by the moderately experienced teaching assistant.

5.2.5 <u>Complexity of Program Decision Structure</u>

As noted earlier in section 3.3 there is some reason to be skeptical about the validity of the McCabe metric values reported by the DPL teaching assistants. While comparisons between the DPL section and the Pascal sections might not be valid, comparisons between the two Pascal sections and comparisons within each of the Pascal sections should still be useful. As can be seen from Figure 14, very little variation exists in the complexity of program decision structure between the Apple and batch Pascal sections. (Note how small the standard error bars are.)

Little variation is not unexpected, however: The problems are too simple and constrained to admit of widely different solutions, particularly with simplistic data structures strongly implied by the problems; and hints from the various program consultants also play a homogenizing role.

Furthermore, little variation exists between the McCabe metrics for programs of the consistent subset and the McCabe metrics for programs of the inconsistent subset. From that observation it may be concluded that errors in programs were caused less by omission of critical program parts or inclusion of hopelessly complicated extraneous code than by incorrect values for variables or comparators. An example of the latter might be executing a loop once more or once

[175] Consult the end-of-semester questionnaire, question 11, Appendix 8.2, for details.

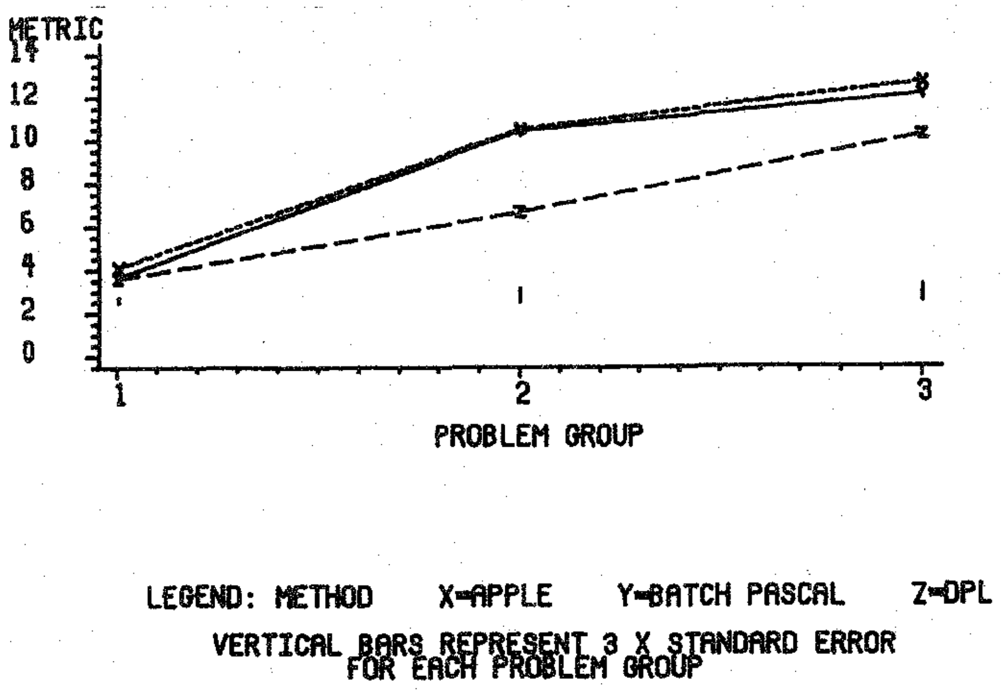

Figure 14: Average McCabe metric for all programs by consistent finishers

less often than was intended.

Another question that occurs is whether program structure deteriorates noticeably as debugging continues, particularly for the Apple section. One might speculate that localized program fixes would corrupt program structure for the on-line group, which generally worked without listings of the latest program version. However, only weak positive correlations (explaining only 22% of the variation) were observed between debugging time and McCabe metric for the consistent subset of the Apple section.[176] Even weaker correlations were observed for the consistent subsets of the other sections. These results follow, almost directly, from the observation that very little variation existed of any kind in the McCabe metric for students in each section. So there was little variation that could be explained by or correlate with some other factor. Deterioration of program structure with debugging time is not ruled out at all, for more complex problems.

---

[176] Higher McCabe metric means more complex program decision structure.

5.2.6 Second-Level Programming Course Follow-Up

Because the experiment was conducted during the fall, 1981 semester, it was possible to informally track the progress of our subjects into the second-level programming course offered in the spring, 1982 semester.[177] There was only one section offered in that course, and the instructor had no knowledge of the section of introductory programming to which each student had belonged. So the observations had no built-in biases. Though the second-level course included some material on assertions, which might have favored the former DPL students, it also included some material on program testing which would have favored the former Pascal groups. More importantly, virtually all the programs in the course had to be written in Pascal, so some estimate could be acquired of whether the former DPL students had gotten up to speed in that language.

Between 22% and 29% of the former students in each section progressed immediately to the second programming course (19 Apple students, 19 batch Pascal, 15 DPL). Fall 1981 introductory programming students comprised 61% of those who finished the second-level course in the spring of 1982.

TABLE 6

Grade Change (in Std. Deviations) from 1st Course to 2nd

	Mean
Apple	-0.18
Batch Pascal	-0.34
DPL	-0.25
Minimum significant group difference	0.12

[177] No attempt was made, however, to collect the same statistics that had been collected from the introductory programming course.

Table 6 shows the grade change, from first programming course to second, as measured in standard deviation units. Since the two first-course instructors and the one second-course instructor each applied different grading criteria, the use of standard deviation units here takes into account a student's performance relative to his peers. In general, since the first-level course weeds out people who have little aptitude for programming, one would expect that the grade performance (in terms of distance above the mean) of a student who continues on to the second course would decrease. That is, an outstanding performer in a large collection of untested beginners will be somewhat less outstanding in a more select second course with classmates of proven potential. If a section's average grade change, in standard deviations, from first course to second was an increase, that would present a strong indication of that methodology's goodness relative to the other instructional methodologies. As Table 6 shows, the Apple section's average grade change was the least negative, a statistically significant amount better than the batch Pascal section's.

Final course grade, to be sure, measures other things besides simple programming ability. Overall, however, the analysis of grade change fits the general pattern of results presented earlier: a mild advantage for the DPL section in relation to the batch Pascal section, and a much stronger advantage for the Apple section in relation to the batch Pascal section. In addition, the grade change data (incorporating second-course grades) suggest that the disappointing performance of the DPL students in Pascal programming at the end of the introductory course was due to inadequate time to learn the new material and not inability to do so.

5.2.7 <u>Subjects' Biographical Factors and Performance</u>

Subject characteristics, as represented on the second day of class, have already been summarized.[178] The characteristics of the subject population at the second day are now compared with the characteristics of course finishers and dropouts.

The characteristics of finishers and dropouts were very similar across the three subject sections.[179] The most

[178] See section 2.5.1.

[179] The differentially higher rate of good experience with logic among the dropouts had only one plausible explanation: Some students had elected to take a non-technical logic course from the philosophy department as an alternative way of fulfilling their mathematics

TABLE 7

Subject Characteristics

	Day 2	Finishers	Dropouts
NUMBER OF SUBJECTS	274	215	59
SEX			
Male	53%	52%	54%
Female	47%	48%	46%
MAJOR			
Computer Science	21%	23%	11%
Mathematics	16%	16%	18%
Other	63%	61%	71%
YEAR			
Freshman	12%	11%	15%
Sophomore	33%	37%	16%
Junior	26%	25%	33%
Senior	21%	21%	24%
Graduate student	5%	4%	7%
Evening college	3%	2%	6%
GPA	2.95	2.98	2.83
SAT VERBAL	536	542	512
SAT MATH	589	600	542
* Good experience with			
Calculus 1	77%	83%	55%
Calculus 2	41%	48%	15%
Logic	13%	12%	20%
Writing	89%	90%	87%

* "Good experience with" means that the subject completed the course in college or high school with a grade of "C" or better.

noticeable difference between finishers and dropouts was that dropouts had considerably lower SAT math scores and substantially less in the way of positive experience with calculus. Although nothing in the introductory programming course relied directly on calculus, it is evident that lack of mathematical sophistication and inexperience with symbol manipulation placed programming students at a disadvantage. Looked at another way, the same skills and interests that promote good performance in calculus classes would also seem to benefit students of programming.

The lower dropout rate for sophomores and computer science majors is probably coincidental: Computer science majors frequently take the first programming course as first-semester sophomores, and evidently were more likely to stick out the course than non-majors. However, computer science majors performed no better in the course than students with other majors. One may conclude that students select computer science as a major more on the basis of career opportunities and interest in the subject than demonstrated aptitude for the discipline.

Figures on dropout rates are presented in Table 8. No special importance is placed on the dropout figures. Apple students tended to drop out sooner, perhaps in response to the early demands of learning to use the microcomputer and its associated software and peripherals. DPL students tended to drop later, perhaps in response to added intellectual demands of the approach as problems became more difficult, perhaps in frustration with compiler problems. The above figures report only official dropouts, however, and undoubtedly are sensitive to counseling of students by the instructors. De facto dropouts (who usually received "incompletes" or "absents" for final course grades) were included in neither the dropout rates reported above, nor the membership of the course finishers' subset used for the other analyses.

In regard to correlations of biographical variables with observed performance measures, only a handful of correlations were of real interest in explaining the experimental results. The figures in Tables 9, 10, and 11 relate only to correlations on the consistent subset, those finishers who achieved consistent success throughout the semester applying the methodologies of their section. The performance measures utilized in the correlations were average measures for each student on the entire semester's problems.

Grade point average was highly correlated (Table 9) with percentage of correct solutions for both batch processing sections, which is what one would expect.

---

requirement. Thus, those students probably had less mathematical aptitude than would appear to be likely at first glance.

TABLE 8

Dropout Rates

Attended first day of class but dropped by end of second week of classes:

Apple	18
Batch Pascal	15
DPI	18

Dropped between end of second week of classes and end of sixth week of classes:

Apple	15
Batch Pascal	11
DPI	5

Dropped between end of sixth week of classes and end of the semester:

Apple	8
Batch Pascal	1
DPI	11

TABLE 9

Correlation of GPA with Percentage Correct Solutions

	Apple	Batch Pascal	DPL
Correlation coefficient	0.12	0.51	0.45
Number of subjects	(45)	(49)	(39)
Significance level	P=0.205	P<0.001	P=0.002

However, no correlation existed whatever for the Apple section, a counterintuitive result.[180]

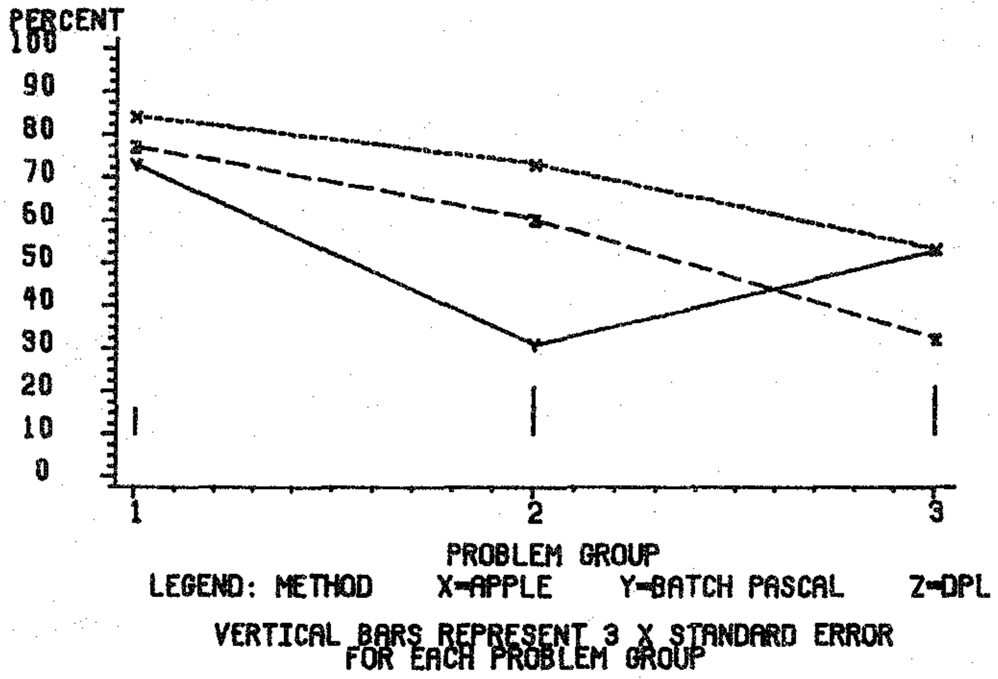

Figure 15: Average percentage entirely correct solutions for all finishers in each section with below-median GPA

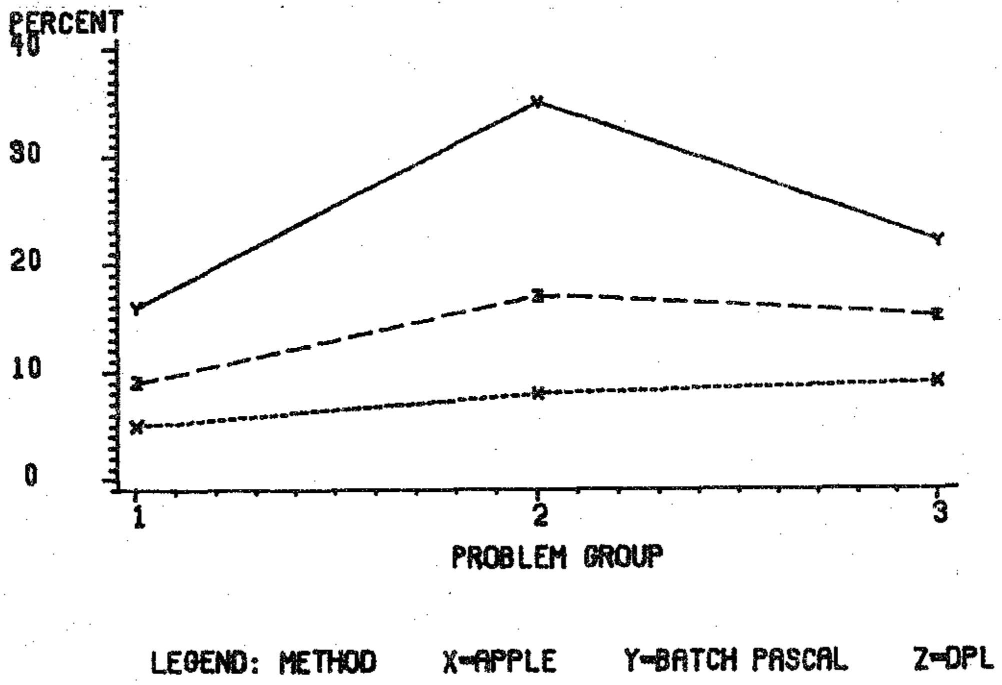

Figure 16: Difference in average percentage correctness between high and low GPA finishers

Figures 15 and 16 also reveal something of the relationship between grade point average and percentage of correct solutions. Figure 15 demonstrates that among those course finishers whose collegiate grade point average ranked in the bottom half of their section, the Apple people impressively outperformed the batch people for most of the course. Not only was the performance of the lower GPA Apple people better than that of lower GPA students in other sections. As Figure 16 shows, their performance was also consistently close to the performance of higher GPA Apple students, coming up short by less than 8% at each problem group. Substantial differences exist in the performance of higher and lower GPA students in both batch sections, but not the Apple section.

What these analyses suggest is that the Apple microcomputer helps marginal students improve their programming performance. A low grade point average may indicate that someone is an undisciplined student, not necessarily that he is unintelligent. The Apple Pascal system evidently concentrates thought and holds attention to the extent that the undisciplined student can work better than he would ordinarily be able to do.

A second set of correlations of interest were those relating SAT scores (as measures of aptitude) to performance measures. There, the only result of interest, for the consistent subset, was that the brighter DPL students were quicker, too. See Table 10 below. SAT math scores were highly negatively correlated with both time before the first run and time after the first run.

Finally, in regard to the relationship between time spent developing the program and subsequent outcomes of run attempts, an interesting correlation exists only for the Apple group. See Table 11.
This result mildly suggests that inadequate preparation time for Apple students led to a higher number of runs with unintended results.

As noted previously, the performance of computer science majors was not significantly different than the performance of non-majors in any of the sections. In fact, the relative advantage enjoyed by the Apple students and by the DPL students, as reported earlier, showed up consistently regardless of which subject subset (whether high or low grade point average, high or low math aptitude, or whatever) was selected for individual analysis.

---

180 The "significance level" referred to in the table refers to the probability that the observed correlation was due to chance rather than experimental treatment. A significance level of .01 would be a conservative significance level for correlations.

TABLE 10

Correlation of SAT Math Scores with Time Measures

	DPL Students' Time Before First Run	DPL Students' Time After First Run
Correlation coefficient	-0.4539	-0.4854
Number of students	(34)	(28)
Significance level	P=0.004	P=0.004

TABLE 11

Correlation of Time Before 1st Run and Unintended Results

	Apple Students
Correlation coefficient	-0.2851
Number of students	(46)
Significance level	P=0.027

5.2.8 Student Feelings About Each Approach

Some people have advocated a switchover from batch processing to microcomputer access on the basis that students would simply find it more fun to use the micros. That expectation was not borne out by the end-of-semester questionnaire. About equal numbers of students in the two Pascal sections found the course to be "enjoyable" or "great fun", somewhat more than in the DPL section.[181] Markedly more students in the Apple section felt satisfied with what

[181] Nearly identical numbers of students in all three sections found the course to be "satisfactory", "enjoyable", or "great fun". See end-of-semester questionnaire, question five, in Appendix 8.2.

they had learned, though all students expressed substantial satisfaction with their experiences.

Chief among the complaints heard from the DPL students were that learning an additional language was a heavy burden, particularly when its introduction was scheduled late in the semester, and that DPL was too restrictive and artificial, when compared to Pascal, particularly as regards output capabilities. In fact, the DPL students preferred to program in Pascal by a substantial margin, 64% to 35%, an impression probably influenced by reliability of the Waterloo compiler and anticipation of future usefulness as much as by particular Pascal features.

The following were typical comments offered anonymously by DPL students at the end of the semester:

> "Although I found this course very challenging, I feel I learned a lot, not only about programming, but about thinking in a logical and orderly fashion."

> "Personally, I found the DPL language to be an exercise in futility, but I see that to those who have difficulty grasping the concept of computer programming, DPL is simple and straightforward enough to be a good teaching language (except, of course, for the totally horrendous mnemonics)."

> "DPL is a primitive language to begin with and it makes me extremely mad that we had to learn it."

> " ... I found DPL to be rather cumbersome when we could have been programming with Pascal."

> "In general, this was a very good course. I learned a lot and I learned to think about the 'correctness of programs'."

> "The DPL approach was very helpful. I'm not sure I would have understood the theory of programming as well if the approach of Conway, Gries, and Zimmerman's *Primer on Pascal* was used."

> " ... As for the DPL experiment, I generally liked the language because of the correctness emphasis. Too bad about the bugs... "

" ... I've been so frustrated at times that I've been ready to pull my hair out. But I must say, overall, I have learned more and found this course more rewarding than any other I've taken."

5.2.9 Conclusions

The DPL approach, which combines methodology and language, seems to have much to recommend itself in practice, as well as in theory. However, it became clear that goodness of language implementation can be as important as the language itself. In this regard, the Apple Pascal system would appear to win hands down.

Experimental results generally supported the hypothesis that the DPL approach would offer significant improvements over the conventional approach. That the results were not even more positive might be explained in several ways. First, it appears that learning to effectively argue program correctness requires skills at least as complex as learning to program, itself. It is easier to verify that one's program produces correct output values than to verify that one has a flawless correctness argument.

Second, the DPL approach requires greater teaching effort: Students generally come into introductory programming with little background in formal logic and critical thinking; they must be taught about the language in which to express assertions about programs, in addition to being taught the programming language proper. Furthermore, correctness arguments must be reviewed by graders and commented upon in addition to the student programs. Where teaching assistants' time for grading and consulting is necessarily limited, the ratio of help delivered to help needed will probably be less for a DPL-like approach than for a conventional approach.

Third, what little experimental evidence exists[182] would seem to suggest that it is easier to learn a more restrictive language after a more flexible one, than the other way around (the way in which this study was conducted). In other words, there might be more advantage to exposure to Pascal first, then DPL, rather than the other way around. Furthermore, one would assume that some

[182] See, for example, G.E. Newton and J.D. Starkey, "Teaching Both PL/1 and FORTRAN to Beginners," SIGCSE Bulletin, Volume 8, Number 3, September, 1976, pp. 106-107.

Also see R.N. Chanon, "An Experiment with an Introductory Course in Computer Science," SIGCSE Bulletin, Volume 9, Number 3, August, 1977, pp. 39-42.

generalized language interference effects undoubtedly occurred when the DPL students were studying Pascal. (For example, DPL and Pascal use different syntactic means to delimit compound statement groupings, a source of some confusion.) Little is known, on a quantifiable basis, about how long to expect a student to take in learning a second language and the conditions under which that learning may be speeded up.

Subjectively, the assertion-based correctness argument approach to program development offers substantial benefits in comparison to the conventional approach, regardless of which programming language is used to implement the designed program. If a program correctness argument seems weak or invalid, the student will realize that the program's logic, itself, may be flawed. Forcing the student to make that argument will expose what he does not know or is unsure about at soon enough to do him some good. It is recommended that the aforementioned approach to program development be combined with attention to program testing after the program has been coded. Consideration of the test data on which the program needs to run should properly occur when formulating postcondition and weakest precondition assertions about the program during its development. Verification that the program actually does run on that input data should take place after the program has been coded.

Advantages and disadvantages of DPL as an actual programming language are discussed in the next chapter.

Microcomputer-based instruction would appear to offer such a large advantage over batch processing, that all other pedagogical techniques being held equal, a significant improvement in instruction can be attained by making that change alone. Furthermore, it should be recalled that the semester in which this experiment was conducted was the first in which the microcomputers were used. Refinement of strategies involving their usage might yield even larger improvements.

Such microcomputer systems represent a financially feasible alternative to a large-scale batch processing facility to support introductory instruction.[183] A

[183] The cost of the Apple microcomputer configuration used in the experiment was approximately $2433 per station, broken down as follows:

Apple II+ (32K memory)	$1058
Pascal language card	144
12" monochrome monitor	237
Serial interface card (for interface to university printer)	128
First disk drive, with controller, cable, and DOS 3.3	477
Second disk drive, with cable	389

scheduling mechanism, just restrictive enough to force students to adequately prepare for their sessions, is strongly recommended. So is the placement, in close proximity to student workstations, of a durable line printer capable of producing hard copy listings at a moment's notice.

Would a DPL implementation designed for a microcomputer system offer the best situation of all? That seems unlikely, in light of comments made in the next chapter, without substantial redesign of the input/output mechanisms to permit labelling of output, modification of the array mechanism, and addition of other language features such as procedures and type definition capabilities. Since one of the goals of an programming course often is to familiarize students with language and program structures that occur in many general purpose languages, it might still be necessary to augment DPL study with instruction in a more general language such as Pascal. However, the students in this study have clearly communicated to us that learning multiple languages in a compressed time frame has its own undesirable consequences.

Each station supported at least 10 students.

Chapter VI

REFLECTIONS ON DPL AS A PROGRAMMING LANGUAGE

It is fair to say that the merits of a programming language may only be appreciated when it has been used extensively for program development. After some intensive experience in developing programs and teaching programming on the only known translator for the language presented in _A Discipline of Programming_, it would seem appropriate to offer some comments on its utility as a program development medium. The reader is assumed, in what follows, to be familiar with Dijkstra's language.[184] Dijkstra's language.

Guarded command set structure

Many, but not all, of the presumed design goals appear to have been met by the language's structure. For example, the guarded command set structure that unifies alternative and repetitive statement types forces the programmer to state explicitly the conditions under which each guarded command group should be executed. This requirement can easily be viewed as desirable for advanced programmers as well as beginners.[185]

The non-determinacy of guarded command selection frees the programmer from artificial constraints in two ways: no longer must an input condition be assigned to one guard when it more reasonably belongs in the overlap of two guards. To wit, the absolute value calculation

```
IF X>=0 -> ABSOLUTE:=X
 | X<=0 -> ABSOLUTE:=-X
FI
```

[184] The locally implemented version of Dijkstra's language is described in P. Koltun, "DPL User's Manual," Technical Report TR82-004, Department of Computer Science, University of North Carolina, Chapel Hill, 1982.

[185] See, for example, M.E. Sime, T.R.G. Green, and D.J. Guest, "Scope Marking in Computer Conditionals -- A Psychological Evaluation," which reports that attaching taxon information to conditionally-executable commands improves the programmer's facility in using such commands.

expresses the symmetry of the guarded execution nicely, without artificial assignment of the X=0 possibility to only one of the guards, as in the usual

```
    IF X>=0 THEN ABSOLUTE:=X
            ELSE ABSOLUTE:=-X
```

Secondly, non-determinacy frees the programmer from explicitly specifying an order in which guards are to be evaluated, where no logical reason for ordering exists. In truth, though, the number of situations in which this flexibility proves advantageous is vanishingly small, at least in short examples presented for introductory instruction.

Permitting multiple guards within a repetitive statement is a nice innovation facilitating concise, consistent presentation of algorithms. Merging of two already-sorted lists, for example, can be expressed very neatly with such a mechanism.

<u>Scope rules and the variable initialization statement</u>

Dijkstra's scope rules reinforce his ideas about separation of concerns, or information hiding, in programming. The mechanisms for explicit inheritance of program variables proved relatively simple for students to pick up. A lecture on variable scope was delivered at the fourth session to introductory students and was understood easily.

The syntactically-distinguished initialization statement for simple variables permits emphasis, in reasoning about program correctness, on starting variables off, at least, with the correct values. Just as requiring explicit variable type declarations has come to be viewed as desirable by most programming language designers, so should requiring (by syntactic mechanisms) special emphasis on variable initialization.

More importantly, variable initialization calls attention to issues of scope in an interesting way. It became clear, in writing a DPL program, that supplying a "dummy" initial value for a variable just to get the initialization requirement out of the way was a signal that the variable's scope was being misconceived. For example, if an integer variable X repeatedly was to be assigned a value from the input, then manipulated and finally assigned to an integer array A, a student's first program version might be

```
.
.
.
DO input remains -> X,IINPUT:LOPOP;
                    manipulations on X;
                    A:HIEXT(X)
OD
```

which would produce the error message that X had not been initialized. (The scope statements for X, A, and IINPUT and the initialization for array A have been intentionally omitted.) So the student might insert the initialization as follows:

```
DO input remains -> X VIR INT,IINPUT:LOPOP;
                    manipulations on X;
                    A:HIEXT(X)
OD
```

Now an error message would be generated to the effect that a variable cannot be repeatedly initialized. (A variable can only be a VIRgin variable once!) Next, thinking that the place to initialize X is before entry to the loop, the student starts to think of what value to give it initially. No value makes more sense than any other, so a dummy value 0 might be used:

```
X VIR INT:=0;
DO input remains -> X,IINPUT:LOPOP;
                    manipulations on X;
                    A:HIEXT(X)
OD
```

But the assignment of a meaningless initial value is a sure tip-off that the scope of X has been misconceived: Since X is used only within the DO-OD repetition, its scope should be that guarded command. The proper implementation is given below, with scope specifications explicitly included.

```
DO IINPUT.DOM>0 -> BEGIN
                     GLOVAR A,IINPUT; PRIVAR X;
                     X VIR INT,IINPUT:LOPOP;
                     manipulations on X;
                     A:HIEXT(X)
                   END
OD
```

where the scope of A (as well as IINPUT) is explicitly inherited from the enclosing context, but X is private to the BEGIN-END program unit, which makes sense because X is used only to store a value between the time it is removed from the integer input and the time it is inserted into array A. So each repetition of the loop requires a new instantiation of private variable X, with its consequent initialization.

<u>Abstract treatment of input/output</u>

Another very nice feature of the locally-designed implementation of Dijkstra's language is the treatment of input and output data collections as arrays, consistent with the treatment of all other arrays in the language.[186] Thus, students did not have to learn specialized formats for input and output statements that had no other application in the language. In fact, the students made very few errors of any kind in input/output usage in DPL.

More importantly, the implementation illustrated to students the concept of abstraction, in a very strong way: The DPL form of input/output emphasized abstraction away from the processing peculiarities imposed by physical unit record devices. The imposition, for example, that input values may only be read once and may only be read from "left" to "right" is a device-dependent restriction that, unfortunately, finds its way into too many languages. The DPL array access mechanism, which treats arrays as double-ended queues for purposes of insertions and deletions, permitted students to formulate algorithms in a more abstract way than would normally be possible. Furthermore, the mechanism avoids peculiarities such as

```
    read (X);
    while not end-of-file do
      begin
        ... processing ... ;
        read (X)
      end
```

wherein you have to read ahead just in order to discover you didn't really want to read that last time, at all. The DPL mechanism permits you to "peer in" and see how many input values are left by examining the current extent of the input domain:[187]

```
    DO INPUT.DOM>0 ->  read (X);
                         ... processing ...
    OD
```

---

[186] The decision to have separate collections for both input and output integers, characters, and booleans, was unfortunate in the sense that it precluded labelled output. However, the separation did emphasize data typing issues.

[187] Pascal lets you look ahead, but only to examine the next input character

Additionally, any actions that would be triggered upon recognition of the nth case from the end of input are now easy to sequence.

The view of arrays as functions

Dijkstra's view of arrays as functions, that is, as total mappings from a domain of subscripts to a range of values, was introduced in A Discipline of Programming, and recapitulated in The Science of Programming, by Gries.[188] While that view presents a conceptually clear picture of arrays to the programmer and facilitates reasoning about subscripts in relation to the array domain, that view is also underexploited in the language Dijkstra formulated and in the exposition of both books. An excuse can be made for Dijkstra's treatment in terms of *his* limited intent, if not for Gries' text. Arrays (single-dimension arrays, at that) unfortunately are discussed with only integer domains, thereby severely limiting the generality and usefulness of the view of arrays as functions. Permitting subscript values to be characters, or for that matter, elements of any ordered set, makes much more sense.

The solution to many problems can be neatly expressed if one views arrays as descriptions of such a mapping. For example, computing a frequency table for the occurrence of alphabetic characters in some text can be viewed as specifying a mapping from letters of the alphabet to integers. Similarly, consider a program which is to read pairs of (possibly unordered) coordinate values and produce a scatterplot of the points they represent. The program might be developed by first assigning elements of a two-dimensional array a print-character or a blank, and then printing the contents of the array on the lines and columns of the output paper. In other words, the array describes a mapping from the cartesian product of the set of rows and the set of columns of the array (corresponding to the possible lines and columns of the output paper) to the set {print-character, blank}:

```
  G: {Rows} X {Columns} --> {print-character, blank}

  G(row i, column j) := { print-character, if the pair (i,j)
                                               occurred in
                                               the input

                          blank,           otherwise       }
```

So the program must first achieve the above assignments to array elements, then simply print out the array's contents, one row per output line.

---

188 D. Gries, The Science of Programming.

One can easily imagine the concept of an initialization mapping for arrays, and perhaps of an inverse mapping[189] as well. Both would facilitate abstractions about programs that manipulate arrays.

<u>Awkwardness of array initialization</u>

In point of fact, the array initialization mechanism represents the major failing in the design of Dijkstra's notation, as borne out from experience in instructing beginners. Since no point in an array's domain should lack an associated value, in Dijkstra's view, the array elements are collected in the form of a double-ended queue for which consecutive domain points all have values. The initialization statement incorporates specification of the initial mapping, by <u>requiring</u> the low bound of the domain and <u>permitting</u> range values to be specified and thus associated with domain values sequencing upward from the low bound.

However, many algorithms admit of no predetermined initial mapping with constants; the array's initial values may come from the (unseen) input, for example. However, a literal domain low bound is required by the initialization, even if all further manipulations in the algorithm utilize only the array domain operators, "array.lob" and "array.hib". Therefore, the programmer is led to specifying a meaningless dummy low bound, just to satisfy the syntactic requirements, which, as was seen before, would be a sure sign of misconceiving the algorithm if it weren't, in this case, a sign of a misconceived language structure. So the programmer specifies

```
A VIR INT ARRAY:=(0)
```

just to get the compiler off his back, which isn't so bad for the experience programmer who understands the necessity. But the novice, the programming innocent, has just been led astray into thinking he can now just as easily refer to 0 as the lowest subscript as to A.LOB, or to A(0) as the lowest element instead of A.LOW, and problems eventually ensue.[190]

---

[189] Think, for instance, of a data base management system. The creation of the data base would require a function that associates keys with attributes:

```
f: {keys} -> {attributes}
```

The retrieval mechanism

```
g: {attributes} -> {keys}
```

which seeks keys with certain attributes, might be regarded as the inverse mapping of f.

In particular, Dijkstra's arrays may grow or shrink at either end. Though 0 may have been the array's lowest subscript initially, that may no longer be true at some later point in the algorithm. Thus, the student's first encounter with concretion has been imposed by the language, when it was abstraction that was to be encouraged. A useful alternative to the single-statement array initialization might be a multi-statement initialization mapping, as suggested before, with a syntactically-distinguished form.

---

190 Very few situations will arise where it is preferable to utilize absolute rather than relative subscript values, when the constraint is imposed that no points in the array domain shall lack an associated range value. An absolute subscript might prove useful as a "key" to refer to an element in a sparse matrix, but not here; in effect, only the non-singular elements of a sparse matrix would even be inserted in Dijkstra's array.

Chapter VII

SUGGESTIONS FOR WOULD-BE EXPERIMENTERS

A review of computer science literature on methodological considerations in performing experiments has already been presented in section 2.6. The comments included here are directed at computer scientists planning to engage in experiments involving human subjects. While the suggestions might amuse the experienced social science investigator, it is hoped that the comments will prove useful to a novice computer science investigator.

The computer scientist's first inclination, after reading a study such as this one, might be to start collecting statistics of his own, and see what interesting relationships emerge among the data. In this field, however, as in others, a power-generality tradeoff exists: the more specific a hypothesis can be formulated, the more powerful an experiment can be designed to test that hypothesis and the more relevant a set of experimental variables can be measured. "Fishing expeditions," in which only generally applicable measures of programmer behavior or learning are collected, probably will not yield sufficiently relevant data to support useful conclusions.

Therefore, the first step for a would-be experimenter would be to think about the phenomenon in which he is interested. For instance, the subject of interest might be how one acquires an appropriate model of computation or how one acquires a knowledge of language rules sufficient to permit discriminations and generalizations about syntactic forms.

The second step should be to pin down, as specifically as possible, hypotheses related to the phenomenon of interest. For example, a general statement that reading programs should be of benefit to a programming student trying to learn a language might be narrowed down to the hypothesis that time spent in reading programs written, say, in Pascal would enable a student to commit fewer syntax errors on a related Pascal programming task than a student who spent an equivalent amount of time writing related Pascal programs.

The third step would be to think about relevant experimental variables, how they relate to the underlying phenomenon, whether they are directly measurable and if not, how they can be approximated. For instance, what is the subject's reading comprehension for prose and how does it relate to his reading comprehension for symbolic language? Can a subject's reading comprehension for symbolic language be improved with practice? What is the subject's information processing ability to discriminate between similar visual situations or generalize abstract rules from similar visual examples? After that, the experimenter can think about how to measure his experimental variables, what the experimental treatment would consist of and what control group, if any, might be used, and what criteria would be used for evaluating group differences.

Clearly, a treatise on experimental design is beyond the scope of this dissertation or the competencies of its author. But starting one's thinking in the right place, as above, should help guide the computer science experimenter along the right track. In particular, becoming familiar with previous work in the area of software psychology can alert the experimenter to the idea that a variety of experimental designs may be required to study different types of phenomena. For instance, the work described in this dissertation falls under the category of "quasi-experimentation."[191] Studies involving computer-user interfaces might utilize classical experimental designs, because they involve observations on only a single programmer and allow for the degree of control required for a true experiment. Exploration of group processes might use the techniques and designs of social psychology. The point is that different problems require different designs. Shneiderman, in his text _Software Psychology_, has some useful comments on experimental design considerations for computer scientists.[192] Expert advice should be sought at an early stage of project planning.

Among the questions that will face the experimenter is the intended scope of an experiment. Looking at the experiment described in this dissertation, one might be led to believe that experimentation over a semester's time with a class of 200 subjects is a preferred situation. Many, many problems relating to difficulty in exerting adequate experimental control exist with such a set-up, however.

The nature of the task involved and the desired generalizability of results should guide decisions regarding scope of experimentation. The subject of this dissertation dealt with learning a complex methodology and language of programming, a task that could not have been accomplished with novices in a short time-period under any circumstances. Furthermore, the goal of the work was evaluating the

---

[191] D.T. Campbell and J.C. Stanley, _Experimental and Quasi-Experimental Designs for Research_, (Skokie, Ill.: Rand McNally, 1966).

[192] Shneiderman, _Software Psychology_.

methodology's applicability to actually teaching semester-long introductory computing courses at the college level. For these two reasons, the scope of this experiment seemed reasonable. In other situations, a limited experiment with the opportunity for tighter experimental control would be preferable. For instance, even in the sample situation described earlier, if one were interested in whether to make program reading an integral part of a course of introductory instruction, the advantages to be gained from reading programs would probably be better explored in experiments of limited scope. The time required for inducing significant subject differences might be measured in hours or days rather than months.

Allowance should always be made for conducting preliminary versions of the experiment, in order to develop materials comprising the experimental treatment, to become familiar with the nature and magnitude of subject differences that might be induced, and to perfect data collection forms and procedures. The experience in this study was that even with a third generation run analysis sheet being used, narrative comments from the subjects were still needed to disambiguate coded subject responses.

Wherever possible, automatable measures ought to be used[193] to capture relevant statistics on-line or to process stored transaction records off-line. If at all feasible, machine-readable forms should be used to capture data that subjects must write down by hand. Interactive programs, with subjects entering needed usage data at a terminal, might be utilized to minimize later data entry.

Plans should be made to archive all relevant source code from programming work to provide a backup in case of necessity. Sad experience can teach that subjects or experiment administrators may not provide data in the form needed. Having an archive to return to for that data can minimize the damage. Plans should also be made to review the quality of data periodically, as it is being collected, so that corrective action may be taken as soon as it appears necessary.

Pains should be taken so that all individuals involved in administering the experiment understand the relevance and importance of data they are asked to provide. The tack taken in this experiment to shield course assistants from the experimental hypotheses in order not to bias the results turned out to be a case of erring on the wrong side: Several assistants failed to collect needed statistics because they thought it wasn't that important.

---

[193] See V.R. Basili and R.W. Reiter, Jr., "Evaluating Automatable Measures of Software Development," <u>Workshop on Quantitative Software Models</u>, IEEE, October, 1979.

Wherever possible, think in advance about how critical statistics might be double checked for validity by independent means. For example, computation center accounting figures might be used to estimate run usage in addition to reports the students provide themselves.

Finally, the role of the experimenter, himself, should be addressed. Wherever possible, the person in charge of the experiment, the principal investigator, should act only as a supervisor, to ensure that experimental procedures are being followed, complete and valid data are being collected, and problems are being dealt with promptly. Avoid involvement in the experiment as a direct participant, as for instance, in teaching one of the sections in the experiment of this dissertation study. Primary responsibilities to students or to other chores may keep the experimenter from performing the supervisory tasks needed to ensure complete, consistent statistics or from filling in when crises arise. The serious illness of one of the teaching assistants in the experiment described earlier, for example, necessitated pinch-hitting by the principal investigator at a critical point in the semester, when time commitments were already stretched thin.

Chapter VIII

APPENDICES

8.1 MID-SEMESTER QUESTIONNAIRE RESULTS

1. As a whole, I am understanding the material in COMP 14:

	Well	Adequately	Poorly
Apple	34%	64%	2%
Batch Pascal	25%	72%	3%
DPL	45%	48%	7%

2. I find COMP 14:

	Exciting	Interesting	Not very Interesting	Boring
Apple	24%	74%	2%	0%
Batch Pascal	15%	79%	6%	0%
DPL	19%	76%	4%	0%

3. I understand the lectures, on the whole,

	Well	Adequately	Poorly
Apple	15%	52%	33%
Batch Pascal	15%	70%	15%
DPL	37%	60%	3%

4. I find the lectures

	Very Helpful	Somewhat Helpful	Not very Helpful	Useless
Apple	8%	53%	34%	5%
Batch Pascal	7%	64%	24%	4%
DPL	24%	61%	13%	2%

5. I find the problem session

	Very Helpful	Somewhat Helpful	Not very Helpful	Useless
Apple	54%	39%	5%	2%
Batch Pascal	33%	45%	20%	2%
DPL	24%	47%	26%	3%

6. I find my problem session instructor's one-on-one tutoring

	Very Helpful	Somewhat Helpful	Not very Helpful	I don't ask for it
Apple	47%	26%	3%	24%
Batch Pascal	30%	32%	13%	26%
DPL	43%	22%	14%	20%

7. I find the clinic instructors (other than my problem session instructor):

	Very Helpful	Somewhat Helpful	Not very Helpful	I don't ask for help
Apple	53%	37%	7%	3%
Batch Pascal	29%	35%	16%	20%
DPL	14%	33%	23%	29%

8. I find the reading assignments:

	Very Clear	Mostly Clear	Seldom Clear	Unclear
Apple	4%	79%	17%	0%
Batch Pascal	11%	70%	19%	0%
DPL	13%	70%	14%	3%

9. I find the reading assignments:

	Very Helpful	Somewhat Helpful	Not very Helpful	I don't do the reading
Apple	21%	66%	10%	3%
Batch Pascal	27%	58%	9%	6%
DPL	42%	51%	7%	0%

10. I find the computer time available to me:

	Adequate	Somewhat Inadequate	Very Inadequate
Apple	47%	29%	24%
Batch Pascal	88%	9%	3%
DPL	80%	14%	6%

11. I find the time this course consumes to be:

	Unreasonable	Very high	Normal	Not much
Apple	19%	67%	14%	0%
Batch Pascal	6%	79%	13%	2%
DPL	13%	61%	25%	0%

8.2 END-OF-SEMESTER QUESTIONNAIRE

(Administered day of final exam)

1. Do you intend to continue with other computer science courses in the future?

	Yes	No	Undecided
Apple	55%	28%	16%
Batch Pascal	51%	27%	22%
DPL	48%	26%	26%

2. Would you recommend enrolling in COMP 14 to a friend?

Possible answers:

Yes (regardless of teaching approach used)
No (regardless of teaching approach used)
Undecided
Depends on teaching approach to be used

	Yes	No	Undecided	Depends
Apple	34%	7%	12%	46%
Batch Pascal	47%	7%	12%	33%
DPL	41%	7%	13%	39%

3. Would you recommend enrolling in COMP 14 to a friend, knowing that the teaching approach to be used was the the one under which you studied this semester?

	Yes	No	Undecided
Apple	72%	18%	10%
Batch Pascal	57%	27%	16%
DPL	36%	49%	14%

4. In general, are you satisfied with what you learned from COMP 14?

	Yes	No	Undecided
Apple	82%	10%	7%
Batch Pascal	70%	15%	15%
DPL	67%	16%	17%

5. Independent of long-term benefits of the course, how enjoyable did you find COMP 14?

	Great fun	Enjoyable	Satis-factory	Somewhat Unpleasant	No fun at all
Apple	10%	33%	27%	19%	10%
Batch Pascal	4%	40%	25%	21%	11%
DPL	7%	26%	38%	20%	9%

6. With respect to the run analysis sheet you were asked to turn in for each problem:

Did you keep track of the requested data as you went along or did you wait until the end to fill in the information?

	Kept track	Waited until end
Apple	25%	66%
Batch Pascal	36%	58%
DPL	35%	65%

How accurate is the information you provided?

Number of runs:	Within 1 run	2 runs	3 runs	5 runs	10 runs
Apple	19%	25%	27%	21%	7%
Batch Pascal	34%	36%	16%	11%	3%
DPL	45%	28%	19%	3%	4%

Number of hours:	Within 1 hr.	2 hrs.	3 hrs.	5 hrs.	10 hrs.
Apple	53%	23%	15%	7%	2%
Batch Pascal	50%	26%	9%	5%	10%
DPL	48%	33%	16%	0%	3%

7. How many times did you go to the computation center (or remote entry station) for a typical assignment?

	once/week	twice/week	once/day	several/day
Apple	1%	66%	7%	12%
Batch Pascal	6%	48%	28%	11%
DPL	4%	46%	17%	19%

8. When you went to the computation center, how long did you typically stay there?

	1-10 min.	10-29 min.	30-59 min.	1-3 hrs.	>3 hrs.
Apple	0%	0%	1%	66%	33%
Batch Pascal	1%	5%	14%	38%	42%
DPL	0%	7%	4%	41%	48%

9. How many run requests did you submit during a typical visit to the computation center?

	1	2	3-5	6-10	>10
Apple	4%	19%	39%	21%	15%
Batch Pascal	1%	10%	51%	28%	10%
DPL	3%	7%	52%	32%	6%

10. Averaged over the last half of the semester, and taking into account that some weeks assignments were due and other weeks assignments were not due, how much out-of-class time did you spend on COMP 14 per week? (Estimate to the nearest hour.)

Apple	11.0 hours
Batch Pascal	13.0 hours **
DPL	12.2 hours

** Includes one figure of 120 hours. The average would be 11.7 hours, excluding that figure.

11. To the best of your recollection, on how many occasions did a program you submitted produce what the teaching assistants or instructor classified as a "bug" in the compiler rather than in your program?

	Never	Once	Twice	>Twice
Apple	52%	22%	12%	13%
Batch Pascal	64%	14%	17%	5%
DPL	36%	35%	17%	12%

12. On how many occasions was a clinic attendant or teaching assistant unable to help you with a problem in your program?

	Never	Once	Twice	>Twice
Apple	18%	15%	12%	55%
Batch Pascal	21%	20%	25%	35%
DPL	26%	25%	13%	36%

13. Toward the end of the semester, did you find the computer time available to you

	Adequate	Somewhat Inadequate	Very Inadequate
Apple	48%	42%	10%
Batch Pascal	44%	46%	10%
DPL	49%	42%	9%

14. Did you find the Conway, Gries, and Zimmerman Pascal text to be

	Very Clear	Mostly Clear	Seldom Clear	Unclear
Apple	7%	63%	24%	6%
Batch Pascal	19%	63%	21%	6%
DPL	6%	54%	22%	19%

15. Estimate the difficulty of each problem from 1 (easy) to 10 (difficult).

Problem	Notes
2 - Conversion to yds, ft, in.	DPL group wrote programs
3 - Armstrong numbers	2-6 in DPL,
4 - Fibonacci numbers	8-9 in Pascal.
5 - Insertion sort inner loop	
6 - Odometer	
8 - Function graphing	
9 - Mean/median	

Problem

	2	3	4	5	6	8	9	avg.
Apple	2.26	3.21	4.13	5.61	6.37	6.79	5.67	4.86
B.P.	2.53	3.28	4.48	6.17	6.91	6.86	5.53	5.21
DPL	2.40	3.28	4.91	5.80	6.89	7.70	7.68	5.52

For the DPL group only

16. In which language do you prefer to program?

DPL	Pascal
35%	65%

17. In which language do you feel it is easier to write correct programs?

DPL	Pascal
42%	58%

8.3 LECTURE SCHEDULES

8.3.1 DPL Lecture Schedule

Lecture	Topic
1	Programs and program correctness
2	Finite state machines: A model of computation
3	Preconditions, postconditions, and boolean algebra
4,5	Variables, initialization and assignment of value, and the notion of scope
6	Order of statement execution and guarded commands
7	Programs using alternative statements
8	Programs using repetitive statements
9	Repetitive processing of input data
10	Algorithm development by stepwise refinement
11	Array variables
12	Using arrays: A searching example
13,14	Finding 1000 prime numbers
15,16	Loops, invariant relations, mathematical induction
17,18	Binary search
19,20	DPL/Pascal differences and similarities
21,22	Pascal data types and data structures
23	Two-dimensional arrays; program modularization
24,25	Procedures, functions, parameter passing, and recursion
26	Considerations beyond program correctness: time-space tradeoffs and optimization hints
27	Recapitulation: programs, variables, and algorithms

8.3.2 Apple and Batch Pascal Lecture Schedules

Lecture	Topic
1......	JCL, programs, and data; reading printouts; and error diagnosis
2......	Program structure and declarations; the program development sequence
3......	Constants, assignments, integer expressions, read, write, and tracing
4......	Selection, conditions, booleans, and boolean expressions
5......	Choosing test data; multiple data set input
6......	Test data selection examples; stepwise refinement example using sorting
7......	Stepwise refinement, quadratic equation example; loops and their implementation
8......	Loops and loop schemata; readln input
9......	Nested loops; output format; real and char data types
10......	Character type, subrange type, and one-dimensional arrays
11......	Arrays, end-of-list conditions, and iteration
12......	Arrays, character strings, arrays of arrays; for loops
13......	Arrays of arrays; sequential search
14......	Insertion sort
15......	Insertion sort with characters; two-dimensional arrays
16......	Subprograms; functions
17......	Procedures and parameter passing
18......	Arrays and subprograms; modularization
19......	Program modularization
20......	Modularization and subprogram testing with drivers

COMP 14 RUN ANALYSIS FORM

Name ______________________

Lab instructor ______________

Date ______________________

Assignment _________________

Total number of runs needed to complete the assignment (Put a mark through the next run number each time you submit a run):

1 2 3 4 5 6 7 8 9 10 11 12
13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35

To the nearest half-hour, record the time expended

prior to 1st run: ________________ hours

after 1st run: ________________ hours

(Count time spent thinking and working on problem, but not time spent waiting in computation center.)

<u>Instructions</u>: Your objective in submitting a run is assumed to be one and only one of the following:

a) running a complete solution to a programming problem through to an <u>entirely satisfactory conclusion</u>;

b) running a partial solution to a programming problem through to an <u>entirely satisfactory conclusion</u>; or

c) discovering how a particular language mechanism works by writing a separate test program.

For each run you make, answer the first question below about whether your objective in submitting the program was completely satisfied, then check off the appropriate categories on the next three questions, as they apply, and briefly describe the changes embodied in each program run after the first, on the reverse side.

Run #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Did the program do what you intended it to?																																			
Yes																																			
No																																			
Your reason for making this run was :																																			
testing a complete solution to a problem																																			
testing a partial solution to a problem																																			
discovering how a language feature works																																			
If the run was not completely satisfactory, characterise its failings or shortcomings:																																			
Error in program logic																																			
Error in programming language usage																																			
Misunderstanding of problem specifications																																			
If the run was not completely satisfactory, how much effort was required to isolate the cause of the problem?																																			
one hour or less																																			
one hour to one day																																			
more than one day																																			
cause never found																																			

FILL OUT REVERSE SIDE, TOO!

DESCRIPTION OF CHANGES

For each program run you submitted, briefly describe the changes made from the previous run, and why the changes were made. Indicate the run number at the left side of the line. Obtain additional sheets if necessary.

Run #

8.5 AN EXAMPLE OF A PROGRAM CORRECTNESS ARGUMENT

Consider this problem: You are a cashier. Write instructions to make change, using the fewest coins possible, for a purchase between 1 cent and 100 cents paid for with a $1 bill. The final situation you want to achieve is that the change you pay out is equal to the difference of 100 cents and the purchase price. The initial situation is that the purchase price is between 1 cent and 100 cents, and that no change has been paid out yet.

Three "algorithms," or specific plans for solution of a problem, are popular for making change: Some people add pennies to the purchase price until the sum is a multiple of 5, then add larger coins until the sum is a multiple of a larger coin denomination, and so on, until the sum is 100 cents; others calculate the change to be given out, then subtract coin values from the change as coins are given out, until the change still to be given is reduced to 0; and still others calculate directly the number of each denomination coin to be returned by dividing the change to be returned by 25 cents to find the number of quarters, subtracting out the value of the returned quarters, and doing the same for dimes, nickels, and pennies.

We now give a program and correctness argument for the second method:

Change reduction algorithm:

Three pieces of information will be maintained, PURCHASE.PRICE, CHANGE.NEEDED, and CHANGE.GIVEN, where PURCHASE.PRICE is the value supplied to the program, and the other two variables receive their initial values in the first two program statements:

```
  CHANGE.NEEDED is 100 - PURCHASE.PRICE
  CHANGE.GIVEN  is 0

  while CHANGE.NEEDED is >= 25, repeat the following:
      dispense a quarter
      reduce the value of CHANGE.NEEDED by 25
      increase the value of CHANGE.GIVEN by 25
      end of repeated instructions.

  while CHANGE.NEEDED is >= 10, repeat the following:
      dispense a dime
      reduce the value of CHANGE.NEEDED by 10
      increase the value of CHANGE.GIVEN by 10
      end of repeated instructions.

  if CHANGE.NEEDED is >= 5  do the following:
      dispense a nickel
      reduce the value of CHANGE.NEEDED by 5
      increase the value of CHANGE.GIVEN by 5

  if CHANGE.NEEDED is >= 1, do the following:
      dispense CHANGE.NEEDED number of pennies
      reduce the value of CHANGE.NEEDED to 0
      increase the value of CHANGE.GIVEN
        by CHANGE.NEEDED
```

Note that when CHANGE.NEEDED has been reduced to less than 10, at most one nickel can be properly dispensed, so no repetitive statement is needed at this stage, contrary to the earlier steps. Also note that when CHANGE.NEEDED has been reduced to less than 5, exactly CHANGE.NEEDED number of pennies can be dispensed directly, since pennies are the smallest denomination coin and constitute the remaining change to be given. The plan described above was chosen to simplify the correctness argument.

Correctness argument: We must argue first that each loop, or repetitive statement, terminates, that is, that the condition in each "while phrase" must eventually become false. Each loop must terminate because CHANGE.NEEDED is positive to begin with (PURCHASE.PRICE between 1 and 100, inclusive, was specified), can only diminish in value, and in fact does diminish in value each time through a loop. So for each loop, if CHANGE.NEEDED is not initially less than the stated coin value in the given loop, eventually it must sink below the stated coin value. Since each loop terminates, in turn, the program terminates when the last loop is over.

Now we must argue that the change dispensed was just what was called for. Were it not for the need to prove this, we might not have used the variable CHANGE.GIVEN. But observe that the sum of CHANGE.NEEDED and CHANGE.GIVEN remains constant (except between the pair of statements changing their values); when CHANGE.GIVEN decreases in value, CHANGE.GIVEN increases by a like amount. CHANGE.NEEDED started out as 100-PURCHASE.PRICE, exactly what we needed to give back; CHANGE.GIVEN started out as 0. Since the sum of the two variables remained constant, when CHANGE.NEEDED falls to 0 (our stopping condition), CHANGE.GIVEN has risen to 100-PURCHASE.PRICE, exactly the change we had to give back. So CHANGE.GIVEN, which records the value of coins dispensed, is exactly what it should be at termination. How do we know we haven't given back more change than we should? If CHANGE.GIVEN were greater than correct, then CHANGE.NEEDED would have to be negative, since their sum is constant. But this is impossible, since each decrement leaves CHANGE.NEEDED nonnegative and the program terminates as soon as CHANGE.NEEDED falls to 0. The program transforms the initial state, when CHANGE.GIVEN was 0 and CHANGE.NEEDED was 100-PURCHASE.PRICE to a final state in which CHANGE.GIVEN was 100-PURCHASE.PRICE and CHANGE.NEEDED was 0.

8.6 EXAMS GIVEN TO DPL AND BATCH PASCAL/APPLE PASCAL STUDENTS

8.6.1 DPL Exam #1

(in seventh week of class)

1. Indicate what is actually or potentially wrong with the syntax or logic of each of the following program segments and give your reason for thinking so. You may assume that portions of the program not shown would be correctly written.

a) The following program segment is intended to calculate and print the Fibonacci numbers less than some limiting value which is known to be greater than 1.

```
FIRST VIR INT:=0;
SECOND VIR INT:=1;
IOUTPUT:HIEXT(FIRST);
IOUTPUT:HIEXT(SECOND);
FIBNUM VIR INT:=FIRST+SECOND;
LIMIT VIR INT,IINPUT:LOPOP;
DO FIBNUM<LIMIT -> IOUTPUT:HIEXT(FIBNUM);
                   FIRST,SECOND:=SECOND,NEXT;
                   FIBNUM:=FIRST+SECOND
OD
```

b) The following program segment is intended to print all Fibonacci numbers less than some value M. That is, if M<=0, no output is to occur; if M=1, the value 0 is to be printed; if M>1, then values 0,1,1, ... are to be printed.

```
IF M<=0 -> SKIP
 | M>0  -> IOUTPUT:HIEXT(0)
 | M>1  -> IOUTPUT:HIEXT(0);
           IOUTPUT:HIEXT(1);
           repetitive statement to generate
           other sequence members
FI
```

c) The input will be a sequence of integer values between 1 and 6. The following program segment is intended to count the number of 1's and 2's in the input.

```
    SINGLES VIR INT:=0;
    DOUBLES VIR INT:=0;
    DO IINPUT.DOM>0 -> NEXT,IINPUT:LOPOP;
                       "ASSUME 'NEXT' IS A NON-VIRGIN
                        INTEGER VARIABLE"
                       IF NEXT=1 -> SINGLES:=SINGLES+1
                        | NEXT=2 -> DOUBLES:=DOUBLES+1
                       FI
    OD
```

2. Given as precondition that the input value to be read may be any integer, characterize what the following program segment does by formulating an appropriate postcondition assertion.

```
    N VIR INT,IINPUT:LOPOP;
    DO N>1000               -> N:=N/10
     | (N>0) AND (N<100) -> N:=N*10
    OD
```

3. Given as precondition that the input is any non-zero integer, write program segments to read a value for a virgin integer variable X, and to establish the truth of the following postcondition, where SIGN is also a virgin integer variable.

(X is positive and SIGN=1) or (X is negative and SIGN=-1)

4. The weakest precondition of a program and a postcondition is the most inclusive description of the initial state for which the given program terminates establishing the truth of the given postcondition. State the weakest precondition for each of the following program segments and postconditions.

```
a) program:         X:=2*X
   postcondition:   0<X<20

b) program:         X:=X+Y
   postcondition:   X>0
```

5. Observe that the MOD operator can be used to examine whether one integer is a multiple of another. That is, for integer variables A and B, if (A MOD B)=0 then A is a multiple of B. Use this understanding of the MOD operator, as well as your other programming knowledge, to fill in the statement or statements needed in each of the blanks in the following program segment, which is intended to print all the positive multiples of 2 less than 1000 and all the positive multiples of 3 less than 1000.

```
"I IS THE NEXT INTEGER TO BE EXAMINED AS A POSSIBLE
 MULTIPLE OF 2 OR 3."
I VIR INT:=1;
DO I<1000 ->
  IF (I MOD 2)=0 OR (I MOD 3)=0      -> _______________
   | ¬((I MOD 2)=0 OR (I MOD 3)=0)   -> _______________
  FI
OD
```

Indicate what you must argue as correct about this program, then briefly do so.

6. Many people have made syntactic mistakes in the usage of semi-colons. To help you appreciate that a semi-colon may only precede the beginning of a statement, answer the following question:

With what DPL symbols or parts of the DPL language may a statement begin? Be specific. (A "part" of the DPL language is something which is written in mixed upper and lower case letters in a syntax diagram, and which is further defined in its own syntax diagram.)

8.6.2 DPL Exam #2

(in eleventh week of class)

1. GRADES is a 1-dimensional integer array of all 10 exam scores for each of the 20 people in a class. Assume that all the exam scores have already been read into the array, in the order: all the scores for student 1, followed by all the scores for student 2, etc.

Write in DPL a program segment to compute and print each student's average exam score. (You may assume that the scores have been properly read into GRADES, but should explicitly provide the output statements for the needed printing of values. You should assume that a lowest subscript value has been specified in the initialization statement for GRADES, but should make no assumption about what that specific lowest subscript value was.)

2. The following program segment is intended to search array A for value X and if the value is found, output the subscript of A at which the value was located. (You should assume that values have already been read into A.)

```
I VIR INT:=A.LOB;
DO A(I)¬=X -> I:=I+1
OD;
IF I<=A.HIB -> IOUTPUT:HIEXT(I)
 | I> A.HIB -> SKIP
FI
```

What is wrong with the program?

Indicate a modification to the program so that it will be correct. (Indicate the change(s) above or re-write the program below, as you choose.)

3. The following program portion reverses the order of the elements of array A, by swapping the outermost remaining values then moving inwards and repeating.

```
"I REPRESENTS HOW MANY TIMES SO FAR A PAIR OF VALUES
 FROM ARRAY A HAS BEEN SWAPPED."
I VIR INT:=0;
DO I<(A.DOM/2) -> A:SWAP(A.LOB+I,A.HIB-I)
                  I:=I+1
 *
OD
```

Formulate an invariant relation that describes, at point *, the portion of the array that remains to be reversed. (The invariant relation should be specified in terms of I and such array domain expressions as are needed.)

4. Sometimes a person reading a program can immediately detect something amiss with the program comments. Seldom do you need to understand the entire program to reach that conclusion. Often the comment says something that makes no sense in terms of English grammar and makes no sense in terms of the programming language's proper usage. Sometimes the problem lies elsewhere.

Refer to the declarations and/or program statements and indicate what is improper with the following comments. (Be as specific as you can.)

```
a) READING VIR INT ARRAY:=(1);
           .
           .
           .
   READING:SWAP(RIGHT,LEFT)              "EXCHANGES VALUES OF
                                          RIGHT AND LEFT"


b) PRIVAR LEFT;                          "LEFT IS THE INDEX
           .                              POSITION 'POINT+1'"
           .
           .
   LEFT VIR INT:=POINT+1;
           .
           .
           .
   DO LEFT<RIGHT -> M:SWAP(LEFT,RIGHT);
                    LEFT:=LEFT+1;
                    RIGHT:=RIGHT-1
   OD
```

```
c) PRIVAR RIGHT        "AN INDEX VARIABLE REPRESENTING
                        AT ONE TIME THE SMALLEST LARGER
                        NUMBER AND LATER IN THE PROGRAM,
                        THE HIBOUND."
```

(Note: See homework problem #6, described in Appendix 8.7, before reading the following exam question.)

5. The following is a correct, commented solution to the odometer problem you worked on. Given as input a sequence of ten unique digits (but not 9 8 7 6 5 4 3 2 1 0), you were to compute the next higher sequence. There are some simple modifications that can be made to the program so that it will compute the next lower sequence, instead. (The input would be any sequence of ten unique digits except 0 1 2 3 4 5 6 7 8 9.) Indicate those modifications in both the programs and comments directly on the program below.

```
"FIND GREATEST INTEGER I SUCH THAT C(I)<C(I+1):"
I VIR INT:=C.HIB-1;
DO C(I)>C(I+1)  -> I:=I-1
OD;

"FIND GREATEST INTEGER J SUCH THAT C(J)>C(I):"
J VIR INT:=C.HIB;
DO C(J)<C(I)  -> J:=J-1
OD;

"SWAP VALUES IN POSITIONS I AND J:"
C:SWAP(I,J);

"REVERSE THE ORDER OF THE VALUES IN POSITIONS I+1
 THROUGH C.HIB.  L AND R ARE THE LEFTMOST AND
 RIGHTMOST POSITIONS AT WHICH VALUES ARE TO BE
 SWAPPED."
L VIR INT:=I+1;
R VIR INT:=C.HIB;
DO L<R -> C:SWAP(L,R);
          L:=L+1;
          R:=R-1
OD
```

8.6.3 DSL Section's Final Exam

(Covers only Pascal material)

1. Pascal permits the definition of additional data types or subranges, beyond the standard set of integers, reals, characters, and booleans. This facility enhances a disciplined approach to programming in at least two ways:

   1. It allows the programmer to delineate for the program's readers the precise set or range of values a variable may take on, thus communicating more about the intended use of that variable than would normally be possible.

   2. It allows the computer to check and notify the programmer when an unintended value is assigned to a variable, thus rendering considerable debugging assistance.

For each of the following brief problem descriptions, indicate the complete variable and/or type declarations you would need for the most important data structures or variables.

a) A program to manipulate variables for hours, minutes, and seconds.

b) A program to tabulate fruit sales, in pounds, for the following fruit: apples, oranges, peaches, lemons, limes, strawberries, tangerines, raspberries, pears, and plums.

c) A program to tabulate the cumulative score for a complete game of bowling.

d) A program to manipulate information about whether each of the 200 parking spaces in a garage is occupied or not.

2. Suppose you are told to write a program segment to determine the range of values in a list of integers to be read in. (The "range", here, is the difference between the largest and smallest values read.) The following three program segments are proposed as solutions. Under certain circumstances (for certain sets of input values) the program segments will produce correct answers.

Identify under which circumstances each program will work. (Think of the circumstances under which each one might not work, then write down the circumstances under which it would work.)

a)
```
MIN:=100000;
MAX:=-100000;
WHILE NOT EOF DO
  BEGIN
    READ(X);
    IF X<MIN THEN MIN:=X
             ELSE IF X>MAX THEN MAX:=X
  END;
RANGE:=MAX-MIN
```

b)
```
READ(X);
MIN:=X;
MAX:=X;
WHILE NOT EOF DO
  BEGIN
    READ(X);
    IF X<MIN THEN MIN:=X
             ELSE IF X>MAX THEN MAX:=X
  END;
RANGE:=MAX-MIN
```

c)
```
READ(X);
MIN:=X;
MAX:=X;
WHILE NOT EOF DO
  BEGIN
    READ(X);
    IF X<MIN THEN MIN:=X;
    IF X>MAX THEN MAX:=X;
    RANGE:=MAX-MIN
  END
```

3. Military people since the days of Julius Caesar have used codes and ciphers to scramble messages and protect their plans against discovery even if the messengers were captured and forced to disclose their messages. A simple-minded cipher would be to change each consonant of the English alphabet into the letter that follows it, and each vowel into the letter that precedes it. (Consider the alphabet as circularly linked: A follows Z and Z precedes A.) Thus the message

ATTACK AT DAWN

would be enciphered as

ZUUZDL ZU EZXO.

If the receiver also knows the enciphering scheme, deciphering the message is no problem. The enciphering, then, is really a mapping which takes a letter from the domain and maps it into some other letter in the range. (Both the domain and range, here, are the alphabet.)

Write a Pascal program to read a message consisting of characters from the input cards and print the coded message enciphered according to the above scheme. Include all declarations, comments, and input/output statements.

5. The following are two program segments to sort a set of N integer values already read into array A. With N=6 and the contents

```
A[1] = 1
A[2] = 6
A[3] = 2
A[4] = 8
A[5] = 31
A[6] = 7
```

state, for each program segment below, the number of times a comparison is made between two array values for each time through the outermost loop (that is, for each new value of I), then add the numbers together to get a total count.

```
a)  (* SELECTION SORT *)
    I:=1;   (* I is the position of the next
               element of A which is to receive
               its proper sorted value *)
    WHILE I<N DO
      BEGIN
        P:=I;  (* P will be the position of the
                  smallest value found so far *)
        J:=J+1;  (* J is the position of the next
                  array value to be compared to
                  the largest found so far *)
        WHILE J<=N DO
          BEGIN
            IF A[J]<A[P] THEN P:=J
                              ELSE;
            J:=J+1
          END;
        (* Swap the values in positions I and P of A *)
        T:=A[I];
        A[I]:=A[P];
        A[P]:=T;
        I:=I+1
      END
```

```
b)  (* INSERTION SORT *)
    I:=2;       (* I is one more than the number of
                   values inserted so far. *)
    WHILE I<=N DO
      BEGIN
        J:=I-1;    (* J is position of next value
                   potentially less than the value
                   to be inserted at this stage  *)
        FOUND:=FALSE;  (* FOUND is truth of "Have
                       found insertion point already" *)
        WHILE (J>=1) AND (NOT FOUND) DO
          BEGIN
            IF A[J]<A[J+1] THEN FOUND:=TRUE
                           ELSE BEGIN
                                  T:=A[J];
                                  A[J]:=A[J+1];
                                  A[J+1]:=T;
                                  J:=J-1
                                END
          END;
        I:=I+1
      END
```

6. For the following Pascal program fragment, assume that the weakest precondition for the program is that X and Y are sorted in increasing order (i.e., no repetitions and X[1]<X[2]<...<X[M] and likewise Y[1]<Y[2]<...<Y[N]. However, there might be some I and J for which X[I]=Y[J].)

The program fragment is to compute an array, U, which contains in ascending order (without repetitions) all elements that are in either X or Y or both.

```
VAR X,Y: ARRAY[1..100] OF INTEGER;
      U: ARRAY[1..200] OF INTEGER;
      M,          (* Actual # of values to be read into X *)
      N,          (* Actual # of values to be read into Y *)
      I,          (* Subscript of next element of X that
                       might be inserted into U            *)
      J,          (* Subscript of next element of Y that
                       might be inserted into U            *)
      K:          (* Subscript of next element of U to
                       receive a value                     *)
        INTEGER;

BEGIN
  READ(M,N);
  statements to read values correctly into X and Y;
  I:=1;
  J:=1;
  K:=1;
  WHILE (I<=M) AND (J<=N) DO
    BEGIN
           IF X[I]<Y[J] THEN BEGIN
                               U[K]:=X[I];
                               K:=K+1; I:=I+1
                             END
      ELSE IF Y[J]<X[I] THEN BEGIN
                               U[K]:=Y[J];
                               K:=K+1; J:=J+1
                             END
      ELSE IF X[I]=Y[J] THEN BEGIN
                               ___________
                               ___________
                             END
      *
    END;
    IF I>M THEN WHILE J<=N DO
                  BEGIN
                    ___________
                    ___________
                  END
           ELSE (* J>N *)
                WHILE I<=M DO
                  BEGIN
                    ___________
                    ___________
                  END
  END.
```

a) Demonstrate your ability to read the above program fragment and construct the needed algorithm by filling in the blanks with the needed statement or statements.

b) If R is the sum of the number of times the first loop is repeated plus the number of times the second loop is repeated plus the number of times the third loop is repeated,

What is the maximum value of R?

What is the minimum value of R?

c) For point *, formulate an invariant relation which describes the contents of U in terms of the contents of X and Y.

d) Offer an argument of the correctness of the completed program. State what must be argued, then do so.

8.6.4 Apple/Batch Pascal Exam #1

(in seventh week of class)

I. Multiple choice. Circle one answer, the best answer, for each question. Read the questions and the answers carefully.

1. A statement group

```
     BEGIN
       2 or more statements
     END
```

  a) Is used for clarity to indicate groups of related statements
  b) Is used to allow many statements to be used where one statement would otherwise be expected
  c) Must appear in an ELSE clause
  d) Is used only to specify the executable part of a program
  e) None of the above

2. The statement READLN(X)

  a) Causes the output device to skip a line after the value of X is read
  b) Causes the value of X to be read and then to be written on a new line
  c) Causes the value of X to be read from a new line
  d) Causes the value of X to be read from the last item on the present line
  e) None of the above

3. A program with the following structure

```
     IF condition THEN GOTO 10;
     1 or more statements;
     20: statement;
     1 or more statements;
     IF condition THEN GOTO 10;
     GOTO 20;
     10: statement;
     1 or more statements
```

  is undesirable because
  a) It cannct operate correctly
  b) It is hard to understand
  c) It has no ELSE clauses
  d) It uses numeric labels
  e) None of the above

4. In an insertion sort of a list of elements, as given in lecture, the first insertion is of

   a) The first element into its ultimate position in the sorted list
   b) The last element into its ultimate position in the sorted list
   c) The second element into its correct position relative to the first element
   d) The smallest element into the first position in the list
   e) None of the above

5. Consider the following program fragment

```
IF X<5
   THEN action1
ELSE IF X<10
   THEN action2
ELSE IF X<15
   THEN action3
ELSE action4
```

   Which statement below is NOT true?
   a) If X holds the value 7, both action2 and action3 will be taken
   b) If X holds the value 10, action3 will be taken
   c) If X holds the value 16, action4 will be taken
   d) If X holds the value -200, action1 will be taken
   e) If X holds the value 15, action4 will be taken

6. A program to read a series of 50 pairs of numbers and to print the sum of the smaller numbers of each pair would involve

   a) A loop within a loop
   b) A loop within a select
   c) A select within a loop
   d) A select within a select
   e) None of the above

7. Which of the following tasks is principally the responsibility of an operating system?

   a) Translate from Pascal to machine language
   b) Produce a listing of the program
   c) Supervise the execution of a program
   d) Control the format of the program's output
   e) Remove the program's guts

8. The program fragment

```
IF A=B
  THEN
  ELSE A:=B+1
```

a) Is illegal because a statement or statement group must follow the keyword THEN
b) Is illegal because A=B should be A:=B
c) Is illegal because A:=B+1 should be A=B+1
d) Is undesirable because it is unclear and should be replaced by
```
   IF NOT (A-B=0)
     THEN A:=B+1
```
e) Is undesirable because it is unclear and should be replaced by
```
   IF A<>B
     THEN A:=B+1
```

9. If a Pascal program includes the declarations

```
CONST
  XXX = 9;

VAR
  I: INTEGER;
```

a) XXX does not require space in the computer's memory during program execution
b) XXX:=I is a legal statement
c) The value of XXX should never be changed between runs of the program
d) The value of XXX may now and then be changed between runs of the program
e) None of the above

10. Given three integers, consider the problem of finding the one whose value lies between the value of the other two. Which of the following would be the best set of test data for this problem?

a) 1, 6, 8
6, 1, 8
8, 1, 6
8, 6, 1

b) 1, 6, 8
4, 7, 10
6, 1, 8
7, 4, 10
8, 1, 6
10, 4, 7
8, 6, 1
10, 7, 4
8, 8, 6
10, 10, 7

c) 8, 1, 6
8, 8, 6

d) 1, 6, 8
6, 1, 8
8, 1, 6
8, 6, 1
8, 8, 6

e) 8, 8, 6
6, 8, 8
8, 6, 8
6, 6, 8
6, 8, 6
8, 6, 6

II. Insert the semicolons that are appropriate in the following Pascal program.

```
PROGRAM TEST(INPUT,OUTPUT)

   LABEL 10

   VAR
     X: INTEGER

   BEGIN
     READ(X)
     (* LOOP *)
        WHILE TRUE DO BEGIN
          IF X<4 THEN GOTO 10
          IF X<10
            THEN X:=X-3
            ELSE BEGIN
                   X:=X-2
                   WRITELN(X)
                 END
          X:=X+1
        END; 10:
     (* END *)
     WRITE ('GOOD BYE')
   END.
```

III. Trace the one of the following Pascal programs under the title corresponding to your section.

<u>Waterloo Section</u>

Assume that the input stream consists of
-1
4

```
PROGRAM TEST(INPUT,OUTPUT);

  LABEL 10;
  VAR
    X: INTEGER;

  BEGIN
    WHILE NOT EOF(INPUT) DO BEGIN
      READLN(X);

      WHILE TRUE DO BEGIN
       IF X<=0 THEN GOTO 10;
        X:=X-2;
        IF (X*X - X)<>0
          THEN WRITELN(19 MOD(X*X-X))
          ELSE WRITELN(0)
      END; 10:

    END
  END.
```

Apple Section

Assume that the file INPUT.TEXT contains

```
-1
4
```

```
PROGRAM TEST(INPUT,OUTPUT);

  LABEL 10;
  VAR
    TXTIN: TEXT;
    X: INTEGER;

  BEGIN
    RESET(TXTIN, 'UNCSYS1:INPUT.TEXT');
    WHILE NOT EOF(TXTIN) DO BEGIN
      READLN(TXTIN,X);

      WHILE TRUE DO BEGIN
        IF X<=0 THEN GOTO 10;
        X:=X-2;
        IF (X*X-X)<>0
          THEN WRITELN(19 MOD X*X-X))
          THEN WRITELN(19 MOD(X*X-X))
          ELSE WRITELN(0)
      END; 10:

    END
  END.
```

8.6.5 <u>Apple/Batch Pascal Exam #2</u>

(in eleventh week of classes)

1. Assume that the constant MAXSIZE has the value 3 and A is declared as ARRAY[1..MAXSIZE] OF INTEGER. Assume for each of the following program fragments that before executing that fragment the input is as follows, where new lines on the page correspond to new lines of the input stream:

```
     1   2   3
     4   5   6
     7   8   9
    10  11  12
    13  14  15
    16  17  18
```

For each fragment give the contents of the array A after the fragment is executed.

a)
```
FOR I:=1 TO MAXSIZE DO
   READLN(A[I])
```

b)
```
FOR I:=1 TO MAXSIZE DO BEGIN
   READ(A[I]);
   READLN
END
```

c)
```
FOR I:=1 TO MAXSIZE DO BEGIN
   READ(A[I]);
   READLN(A[I])
END
```

2. Let A be declared

```
    ARRAY[1..3] OF ARRAY[4..6] OF REAL
```

Assume that the following program fragment has just been executed.

```
    VALUE:=10;
    FOR I:=1 TO 3 DO
       FOR J:=4 TO 6 DO BEGIN
          A[I][J]:=VALUE;
          VALUE:=VALUE+1
       END
```

a) Give the contents of A[2,6].

b) Give the contents of A[3].

3. Trace the following Pascal program which transliterates German sentences, assuming that the input stream holds the three characters JA. . The program is meant to run on a noninteractive computer.

```
PROGRAM TRANSLIT(INPUT,OUTPUT);
  LABEL 10,20;

  CONST
    NUMLETTERS=3;
    MAXLENGTH=20;

  VAR
    GERLETTER,    (* LETTERS TO TRANSLITERATE *)
    ENGLETTER:    (* CORRESPONDING ENGLISH LETTERS *)
      ARRAY[1..NUMLETTERS] OF CHAR;
    SENT: ARRAY[1..MAXLENGTH] OF CHAR;    (* SENTENCE TO
                                            TRANSLITERATE *)
    CHARNUM,  (* CHARACTER NUMBER IN SENTENCE *)
    LETTERNUM,  (* INDEX IN TRANSLITERATION TABLE *)
    SENTLENGTH:  (* SENTENCE LENGTH *)
      0..MAXINT;
```

```
BEGIN
   (* SET CORRESPONDING LETTERS *)
     GERLETTER[1]:='J';
     ENGLETTER[1]:='Y';
     GERLETTER[2]:='V';
     ENGLETTER[2]:='F';
     GERLETTER[3]:='W';
     ENGLETTER[3]:='V';

   (* READ AND ECHO SENTENCE *)
     WRITE('GERMAN SENTENCE: ');
     CHARNUM:=1;
     (* LOOP *)
       WHILE TRUE DO BEGIN
         READ(SENT[CHARNUM]);
         WRITE(SENT[CHARNUM]);
         IF SENT[CHARNUM]='.'
           THEN GOTO 10;
         CHARNUM:=CHARNUM+1
       END; 10:
     (* END *);
     SENTLENGTH:=CHARNUM-1;

   (* TRANSLITERATE AND PRINT SENTENCE *)
     WRITELN;
     WRITE('ENGLISH TRANSLITERATION: ');
     FOR CHARNUM:=1 TO SENTLENGTH DO BEGIN
       (* TRANSLITERATE AND PRINT THIS CHARACTER *)
         LETTERNUM:=1;
         (* LOOP *)
           WHILE TRUE DO BEGIN
             IF LETTERNUM > NUMLETTERS
               THEN GOTO 20;
             IF SENT[CHARNUM]=GERLETTER[LETTERNUM]
               THEN BEGIN
                      SENT[CHARNUM]:=ENGLETTER[LETTERNUM];
                      LETTERNUM:=NUMLETTERS+1
                    END
               ELSE LETTERNUM:=LETTERNUM+1
           END; 20:
         (* END *);
         WRITE(SENT[CHARNUM])
     END;
     WRITE('.')
END.
```

5. Circle the best answer for each of the following multiple choice questions.

a) Assume that A and B are real variables and the assignments

```
A:=0.6
B:=0.26
```

are executed. The condition (A*A=0.1+B) is then computed to have the value FALSE despite the fact that 0.6 times 0.6 is equal to 0.1 + 0.26 because

1) A and B must be specified in the E format
2) The computer does multiplication of reals in different ways at different times
3) Real constants are not allowed in Pascal
4) Real numbers are not represented exactly in the computer
5) None of the above

b) Let A be declared ARRAY[1..20] OF REAL. A program fragment to change to zero all negative elements of A should have as its main control structure

1) a FOR iteration
2) a WHILE loop
3) a FOR iteration nested inside a FOR iteration
4) a WHILE loop nested inside a WHILE loop
5) a FOR iteration nested inside a WHILE loop

c) Assume that all NC automobile license plates have exactly six characters. A list of such license plate 'numbers' would be best stored as

1) a 1-dimensional array of characters
2) a 1-dimensional array of characters plus a 1-dimensional array of integer lengths
3) a 2-dimensional array of characters
4) a 2-dimensional array of characters plus a 1-dimensional array of integer lengths
5) a 2-dimensional array of characters plus a 2-dimensional array of integer lengths

d) If the declaration

```
  A: ARRAY[-5..5] OF -10..10;
```

occurs in the VAR field of a program and the program includes a pair of statements

```
  READ(I);
  A[I]:=I
```

that are executed when the next item in the input stream has the value 8,

1) A syntax error will be detected by the compiler
2) An error will be detected during program execution
3) The program will calculate an incorrect answer
4) The programmer will trip when picking up his listing
5) No errors will result

e) If your program for a COMP 14 programming assignment includes a GOTO statement exiting from a FOR iteration, will lose credit because of the chance that

1) The compiler will find a syntax error
2) The run-time system will detect an error during execution
3) The program will compute an incorrect answer
4) The program will be hard to debug because of a structure with two exits
5) None of the above

8.6.6 Apple/Batch Pascal Final Exam

1. For each of the data types:

a) ARRAY[1..10] OF ARRAY[-2..3] OF REAL
b) ARRAY[2..6] OF CHAR
c) BOOLEAN
d) 'A'..'B'
e) INTEGER

we wish to know which of the following properties apply to variables of that type.

1) The variable can have only two legal values. a b c d e
2) The variable can serve as the index of an array. a b c d e
3) The variable holds at one time a collection of values. a b c d e

Indicate your answers by circling the corresponding letter for each of the types to which the properties apply.

2. Assume that a main program declares INTEGER variables with the names A and B and that during execution it arrives at a statement

```
P(A,B)
```

when its variable A has the value 5 and its variable B has the value 1. Assume that the procedure P is defined by

```
PROCEDURE P(VAR B: INTEGER; C: INTEGER);
   VAR
     A: INTEGER;

   BEGIN
     C:=2*C;
     A:=C-1;
     B:=B+C+A
   END;
```

Give the values of

a) the variables and parameters of P just before the executable part of P is executed
b) the variables and parameters of P just before the procedure P returns to the main program
c) the variables A and B in the main program just after control is returned from the procedure P.

Use "?" if a value is unknown.

3. Assume that you must write a program to generate a concordance for a text string provided as input - it lists all of the words in the text in alphabetical order and for each word it prints the number of times that the word appeared in the input text. The algorithm that you produce assigns values to three arrays: WORDLIST, such that WORDLIST[POS] holds the POSth word in alphabetical order; COUNT, such that COUNT[POS] holds the number of occurrences of the word stored in WORDLIST[POS]; and LENGTH, such that LENGTH[POS] holds the number of characters in the word in WORDLIST[POS]. Here is the algorithm you produce:

```
NUMWORDS:=0
Loop
   Exit if there is no more input;
   Read the next word into WORD and set           (1)
     WORDLENGTH to its length;
   Set POS to the position of the first word      (2)
     in WORDLIST that is greater than or
     equal to (in alphabetical order) WORD;
   IF WORDLIST[POS] is equal to WORD              (3)
     THEN COUNT[POS]:=COUNT[POS]+1
     ELSE BEGIN
            Move all of the words in WORDLIST     (4)
              from positions POS through
              NUMWORDS down one position;
            Move all of the lengths in LENGTH     (5)
              from positions POS through
              NUMWORDS down one position;
            Move all of the numbers in COUNT      (5)
              from positions POS through
              NUMWORDS down one position;
            NUMWORDS:=NUMWORDS + 1;
            Set WORDLIST[POS] to WORD;            (6)
            LENGTH[POS]:=WORDLENGTH;
            COUNT[POS]:=1
          END
End;
Print the list of words in WORDLIST and the       (7)
  number of occurrences of each (from COUNT)
```

Assume that no word will appear that is greater than MAXLENGTH characters long, and no more than MAXWORDS words will be encountered, where MAXLENGTH and MAXWORDS are declared as constants.

a) Give type definitions necessary to allow each of the variables WORD, WORDLIST, COUNT, and LENGTH in this main program to be declared to have a single-word type.

b) Using the types defined in part a, give the declarations required for WORD, WORDLIST, COUNT, and LENGTH in the main program.

c) The numbers 1-7 in the right margin mark lines that specify the actions of seven subprograms that are required:

1. READWORD
2. POSFIND
3. WORDEQUAL
4. SHIFTWORDS
5. SHIFTNUMS
6. WORDASSIGN
7. CONCPRINT

The number 5 appears in the margin twice to indicate two separate invocations of a single subprogram. For each subprogram give a full FUNCTION statement or PROCEDURE statement needed to head the definition of the subprogram. You need not write any part of the subprogram except for the FUNCTION or PROCEDURE statement.

4. Assume that a program includes a procedure beginning with the statement

```
PROCEDURE Q(A: REAL; VAR B: REAL);
```

Assume that the program declares the real variables C, D, and E. Which of the following two invocations would be legal?

```
1) Q(C,D+E)
2) Q(D+E,C)
```

a) Neither
b) Number 1 only
c) Number 2 only
d) Both

5. You are to write a Pascal program that is given as input a string of alphabetic characters terminated by a blank and prints the "next string", where "next string" is defined by thinking of the characters 'A' through 'Z' like digits and thinking of the string like a number made up of digits. Thus the next string after "A" is "B", the next string after "Z" is "AA", the next string after "AB" is "AC", the next string after "AZZ" is "BAA", the next string after "ZZ" is "AAA", and the next string after the blank string is "A".

   You may assume that the input string will be at least one character shorter than the memory you have set aside to hold the string. You will probably need to use the predefined function SUCC; assume that SUCC applied to any letter but 'Z' gives the next letter in the alphabet. Your program may omit a header comment, but it should include all other commenting and formatting that is normally required.

6. Assume that the array LIST is declared ARRAY[1..8] OF INTEGER and that LIST has the following contents:

LIST[1]	2
LIST[2]	5
LIST[3]	6
LIST[4]	8
LIST[5]	9
LIST[6]	12
LIST[7]	14
LIST[8]	?

Assume that KEY and ELT are both declared INTEGER and that we have our choice of the following two algorithms to set ELT to the index of the element of LIST that holds the value in KEY, or to set ELT to zero if there is no such element.

```
Algorithm 1:  LIST[8]:=KEY;
              ELT:=1;
              Loop
                Exit if LIST[ELT]=KEY;
                ELT:=ELT+1
              End;
              IF ELT=8
                 THEN ELT:=0

Algorithm 2:  BEGLIST:=1;
              ENDLIST:=7;
              Loop
                 Exit if ENDLIST-BEGLIST<=0;
                (* FIND THE MIDPOINT OF THE CANDIDATE
                   PART OF THE LIST *)
                   ELT:=(BEGLIST+ENDLIST) DIV 2;
                (* HALVE THE CANDIDATE PART
                     OF THE LIST *)
                   Select
                      IF KEY<LIST[ELT]
                         THEN ENDLIST:=ELT-1
                      ELSE IF KEY>LIST[ELT]
                         THEN BEGLIST:=ELT+1
                      ELSE BEGIN
                             BEGLIST:=ELT;
                             ENDLIST:=ELT
                           END
                   End
              End;
              IF KEY=LIST[BEGLIST]
                 THEN ELT:=BEGLIST
                 ELSE ELT:=0
```

Each of these algorithms has a loop that includes an exit test. Successively for the values

a) 14
b) 17
c) 5

answer the following questions:

With that value in KEY, how many times will the exit test of algorithm 1 be executed? With that value in KEY how many times will the exit test of algorithm 2 be executed?

d) (Extra credit)
If KEY holds a value in some element of LIST, for each algorithm give the maximum number of times its loop's exit test can be executed.

8.7 PROBLEM ASSIGNMENT SUMMARIES

1. Copy and run (different program for each section)

2. Length in yards, feet, and inches.

   Write a program which reads an integer value representing a length, in inches, and converts that value into the equivalent yards, feet, and inches. Print out the original input value, followed by the computed yards, feet, and inches, in that order. Assume that the input value will be between 0 and 32767, inclusive.

3. Armstrong numbers

   An n-digit number is an Armstrong number if the sum of the n th power of the digits is equal to the original number. For example, 371 is a 3-digit Armstrong number because the sum of the 3 rd power, or cube, of each of its digits equals 371. Write a program to read a 3-digit input integer value, print out that same value, and print "True" if the value is an Armstrong number or "False" if the value is not an Armstrong number. The input integer value will be between 100 and 999, inclusive.

4. Fibonacci numbers

   The sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... , in which each number is the sum of the preceding two, is called a Fibonacci number sequence, after the great pre-Renaissance mathematician who discovered it. Write a program which reads an integer input value, prints that same value, then calculates and prints all Fibonacci numbers less than that integer input value. The input value may be any integer.

5. Insertion sort inner loop

   Write a program to read an integer n followed by a list of n integers and then to reorder this list, in place, by leaving the relative order of all but the last item unchanged but inserting the last item somewhere within, before, or after the other items according to the following insertion rule. The last item is to be inserted immediately after the bottommost of the other items which is less than or equal to the item to be inserted. If the item to be inserted is less than all of the other items, the insertion should be before all of these items. Thus, for example, if the first input value is 6 and the remaining inputs are 1, 9, 2, 5, 7, 3, the 3 will be inserted after the 2 producing 1, 9, 2, 3, 5, 7. Note that if the list of all but the element to be inserted is initially in ascending order, the insertion will cause the whole list to end up in ascending order.

Print the resultant list. Assume that the program should work with values of n up to 10.

6. Odometer problem (Dijkstra's Next Permutation Problem)

Given as input an ordered sequence of ten single-digit positive integers representing a mileage reading, where each one of the ten integers is a different number between 0 and 9 inclusive, calculate and print the next higher mileage where, once again, each of the ten digits is a different number between 0 and 9. For example, for input 6 2 9 5 8 3 7 4 1 0, the next higher sequence would be 6 2 9 5 8 4 0 1 3 7. The restrictions on the input sequence are that the ten values will each be a different number between 0 and 9, and that the sequence will not be 9 8 7 6 5 4 3 2 1 0, for which there is not a next higher mileage obeying the given rule.

7. Pattern match (Apple and batch Pascal sections only)

Write a program that will
  a) read a sentence
  b) read a word or phrase
  c) print whether the word or phrase is contained in the sentence (including precisely the same blanks).

7a. Bowling program re-write (DPL section only)

Re-code into Pascal the complete DPL program in the lecture notes to score a game of bowling. (Given an input sequence representing a legal game-full of bowling pin counts, calculate the frame-by-frame cumulative score.)

7b. Letter concordance (DPL section only)

Write a Pascal program to tabulate the number of occurrences of each letter of the alphabet in a given input text.

8. Function graphing (all sections in Pascal)

Write a program to graph a mathematical function, F(X), on a printed page. The function should be specified as a function subprogram, and the program should be structured so that a different function subprogram could be supplied without change to the remainder of the program. Input values specify the minimum and maximum values of the function domain, and the maximum value in the range for the given domain. Output should consist of a graph, with labeled axes, of the curve over the domain.

9. Statistical subroutines

Write a program to read an integer n followed by a sequence of n real values, then one more integer value designating whether to calculate the mean or the median of the real inputs. The subprogram to calculate the median should sort the input values, using the insertion sort algorithm introduced in assignment #5.

8.8 NOTES ON THE MCCABE COMPLEXITY METRIC

The McCabe metric is a measure of cyclomatic complexity of programs. It relates intuitive complexity and graph theoretic complexity. Complexity, as measured by the program metric, depends only on the decision structure of the program.

How to compute it:

Complexity is the number of conditions in the program, plus one.

Notes:

1. A program with no branching at all has complexity 1.
2. A program which calls a subprogram has complexity equal to the number of conditions in the program plus the number of conditions in the subprogram, plus one.
3. A compound predicate C1 AND C2 contributes 2 to complexity (it has two conditions) because it could be regarded as

   ```
   IF C1 THEN IF C2 THEN _____ without using AND.
   ```
4. A case statement with N possible values of the case expression contributes N to the complexity. Hence,

   ```
   CASE ERRNBR OF 1: _________
                  2: _________
                  3,4,5: _____
                  END
   ```

   would contribute 5 to the complexity.
5. For a conditional,

   ```
   IF condition THEN __________     contributes 1.
   IF condition THEN __________
                ELSE __________     contributes 1.
   ```
6. For a repetition,

   ```
   WHILE condition DO _________     contributes 1.
   FOR I:=exp1 TO exp2 DO _____     contributes 1.
   ```

7. For a DPL guarded command, each guard contributes to complexity. Thus,

```
  IF  condition1 -> _______
   | ¬condition1 -> _______
  FI
```

would contribute 2 to complexity.

8.9 DPL COMPILER BUG LIST

1. Serious and unpredictable inability to handle nesting of program units. Adding an outermost program unit to facilitate running a program on multiple sets of test data introduced spurious syntax errors on occasion.

2. Repetitive statement (DO-OD) may not be the first statement (after scope declarations) of a program unit.

3. Serious and unpredictable compiler errors due to register allocation problems, particularly in compound guards of DO-OD and IF-FI constructions.

4. Array operator ALT (in either syntactic form) does not work at all.

5. Scope rules do not work correctly for array variables - arrays are not deactivated at the end of their private scope. (Simple variables are deactivated correctly.)

6. (Legally) altering (from run to run) the order of identifiers in the identifier-list of a scope declaration can cause a spurious error to be identified.

7. The syntactic recognition of array domain operators is incorrectly implemented. To wit, the guard X=(A.HIB-1) is flagged as an error, while X=((A.HIB)-1) is treated properly.

8. Actual syntax errors are flagged in misleading ways. Ex., IF I<J<K -> is flagged as an "internal stack overflow" compiler error; IOUTPUT:HIEXT(arrayname) (attempting to print an array all in one step) is flagged as an unrecognizable error.

9. The compiler, in several instances, outputs voluminous compiler trace diagnostics (acts as though it has found an error), but continues to correctly translate the program, then correctly executes it.

10. Severe, but inconsistent, limits on amount of input that can be supplied to a program, as well as output that can be generated by it. For example, 32 pieces of input data gives compiler error "There is no more space in the register allocator."

11. A half-dozen characters (_,&,#, etc.) are recognizable by the compiler, but have no legal usage in the language other than appearing in comments They may not be used as part of variable names.

12. Unpredictable and unreasonable limits on depth of nesting of DO-OD's and IF-FI's seems to exist. Error message indicates a compiler error, "Internal stack overflowed."

BIBLIOGRAPHY

Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The Design and Analysis of Computer Algorithms, Reading, Mass: Addison-Wesley Publishing Co., 1974.

Anderson, R.B., Proving Programs Correct, New York: John Wiley & Sons, 1979.

Baker, F.T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Volume 11, Number 1, 1972, pp. 56-71.

Baker, A.L., and Zweben, S., "A Comparison of Measures of Control Flow Complexity," IEEE Transactions on Software Engineering, Volume SE-6, Number 6, 1980, pp. 506-512.

Basili, V., and Reiter, R.W., Jr., "An Investigation of Human Factors in Software Development," Computer, Volume 12,12, December, 1979, pp. 21-38.

________., "Evaluating Automatable Measures of Software Development," Workshop on Quantitative Software Models, IEEE, October, 1979.

Bezanson, W.R., "Teaching Structured Programming in FORTRAN with IFTRAN," SIGCSE Bulletin, Volume 7, Number 1, February, 1975, pp. 196-199.

Bishop, J., "The Portable DPL Compiler Project," Master's thesis, Technical Report TR80-008, Department of Computer Science, University of North Carolina, Chapel Hill, 1980.

Bohm, C. and Jacopini, G. "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," Communications of the ACM, Volume 9, Number 5, 1966, pp. 366-371.

Bowles, K., "A CS1 Course Based on Stand-Alone Microcomputers," SIGCSE Bulletin, Volume 10, Number 1, February, 1978, pp. 125-127.

________., Beginner's Guide for the UCSD Pascal System, New York: Byte/McGraw Hill, 1979.

Brooks, F.P., Jr., The Mythical Man-Month, Reading, Mass.: Addison-Wesley, 1975.

Brooks, R., "Toward a Theory of the Cognitive Processes in Computer Programming," International Journal of Man-Machine Studies, Volume 9, 1977, pp. 737-751.

________., "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology," Communications of the ACM, Volume 23, Number 4, 1980, pp. 207-213.

Campbell, D.T. and Stanley, J.C., Experimental and Quasi-Experimental Designs for Research, Skokie, Ill.: Rand McNally, 1966.

Card, S.K., Moran, T.P., and Newell, A., "The Keystroke-Level Model for User Performance Time with Interactive Systems," Communications of the ACM, Volume 23, Number 7, 1980, pp. 396-410.

Chanon, R.N., "An Experiment with an Introductory Course in Computer Science," SIGCSE Bulletin, Volume 9, Number 3, August, 1977, pp. 39-42.

Chapin, N., "A Measure of Software Complexity," AFIPS Conference Proceedings, Volume 48: 1979 National Computer Conference, Montvale, NJ: AFIPS Press, 1979.

Chen, E.T., "Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, Volume SE-4, Number 3, 1978, pp. 187-194.

Cheney, P., "Cognitive Style and Student Programming Ability: An Investigation," AEDS Journal, Summer, 1980, pp. 285-291.

Conway, R., Gries, D., and Zimmerman, E.C., A Primer on Pascal, Cambridge: Winthrop Publishers, 1976.

Curtis, B., Sheppard, S.B., and Milliman, P., "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Proceedings of the Fourth International Conference on Software Engineering, New York: IEEE, 1979.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., and Love, T., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, Volume SE-5, Number 2, 1979, pp. 96-104.

Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R., Structured Programming, New York: Academic Press, 1972.

Daly, C., Embley, D., and Nagy, G., "A Progress Report on Teaching Programming to Business Students Without Lectures," SIGCSE Bulletin, Volume 11, Number 1, February, 1979, pp. 247-251.

de Groot, A.D., Thought and Choice in Chess, New York: Basic Books, Inc., 1965.

DeMillo, R.A., Lipton, R.J., and Perlis, A.J., "Social Processes and Proofs of Theorems and Programs," Communications of the ACM, Volume 22, Number 5, 1979, pp. 271-280.

Dijkstra, E.W., "A Constructive Approach to the Problem of Program Correctness," BIT, Volume 8,3, 1968, pp. 174-186.

________., "The Structure of the THE Multiprogramming System," Communications of the ACM, Volume 11, Number 5, 1968, pp. 341-346.

________., "Correctness Concerns and, Among Other Things, Why They Are Resented," SIGPLAN Notices, Volume 10, Number 6, June, 1975, pp.546-550.

________., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Communications of the ACM, Volume 18, Number 8, 1975, pp. 453-457.

________., A Discipline of Programming, Englewood Cliffs: Prentice-Hall, 1976.

DiMarco, Bird, and Norton, "Life Style, Learning Style, Learning Structure, Their Congruences and Student Attitudes and Performance in a Data Processing Course," Journal of Educational Data Processing, Volume 16, Number 2, 1979, pp. 1-8.

Elshoff, J.L., "An Analysis of Some Commercial PL/I Programs," IEEE Transactions on Software Engineering, Volume SE-2, Number 2, 1976, pp. 113-120.

________., and M. Marcotty, "On the Use of the Cyclomatic Number to Measure Program Complexity," SIGPLAN Notices, Volume 13, Number 12, 1978, pp. 29-40.

Elspas, B., Levitt, K.N., Waldinger, R.J., and Waksman, A., "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Volume 4, Number 2, 1972, pp. 97-147.

Embley, D.W., and Nagy, G., "Behavioral Aspects of Text Editors," ACM Computing Surveys, Volume 13, Number 1, 1981, pp. 33-70.

Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Volume 15,3, 1976, pp. 182-211.

Fitzsimmons, A., and Love, T., "A Review and Evaluation of Software Science," *ACM Computing Surveys*, Volume 10, Number 1, 1978, pp. 3-18.

Floyd, R.W., "Assigning Meanings to Programs," *Proceedings of Symposia in Applied Mathematics*," Volume 19, American Mathematics Society, 1967, pp. 19-32.

Freund, K., "The Design and Abstract Specification of a Translator Module," Master's thesis, Technical Report TR79-012, Department of Computer Science, University of North Carolina, Chapel Hill, 1979.

Furuta, R., and Kemp, P.M., "Experimental Evaluation of Programming Language Features: Implications for Introductory Programming Languages," *SIGCSE Bulletin*, Volume 11, Number 1, 1979, pp. 18-21.

Gannon, J.D., and Horning, J.J., "Language Design for Programming Reliability," *IEEE Transactions on Software Engineering*, Volume SE-1, Number 2, 1975, pp. 179-191.

Gannon, J.D., "An Experimental Evaluation of Data Type Conventions," *Communications of the ACM*, Volume 20, Number 8, 1977, pp. 584-595.

George, J., "An Abstract Machine as an Aid to Compiler Portability," Master's thesis, Technical Report TR79-017, Department of Computer Science, University of North Carolina, Chapel Hill, 1979.

Gerhart, S.L., "Methods for Teaching Program Verification," *SIGCSE Bulletin*, Volume 7, Number 1, February, 1975, pp. 172-178.

Gould, J.D., and Drongowski, P., "An Exploratory Study of Computer Program Debugging," *Human Factors*, Volume 16,3, 1974, pp. 258-277.

Gould, J.D., "Some Psychological Evidence on How People Debug Computer Programs," *International Journal of Man-Machine Studies*, Volume 7, Number 2, 1975, pp. 151-182.

Gries, D., *The Science of Programming*, New York: Springer-Verlag, 1981.

Halstead, M.H., *Elements of Software Science*, New York: Elsevier North-Holland, 1977.

Hamming, R.W., "A Philosophy for Computer Science or My Prejudices and Confessions," SIGCSE Bulletin, Volume 7, Number 4, 1975, pp. 16-18.

Hanson, A., and Maly, K., "A First Course in Computer Science: What It Should Be and Why," SIGCSE Bulletin, Volume 7, Number 1, February, 1975, pp. 95-101.

Hetzel, W.C., "An Experimental Analysis of Program Verification Methods," (Ph.D Dissertation, University of North Carolina, 1976), Dissertation Abstracts International, Volume 37, 1977, p. 4054B.

Hintzman, D.L., "Effects of Repetition and Exposure Duration on Memory," Journal of Experimental Psychology, Volume 83, Number 3, 1970, pp. 435-444.

Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, Volume 12, Number 10, 1969, pp. 576-583.

Hoare, C.A.R., and Wirth, N., "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica, Volume 2,4, 1973, pp. 335-355.

Hsia, P., and Petry, F.E., "A Framework for Discipline in Programming," IEEE Transactions on Software Engineering, Volume SE-6, Number 2, March, 1980, pp. 226-232.

Jensen, K., and Wirth, N., Pascal User Manual and Report, New York: Springer-Verlag, 1975.

Jones, G.A., and Walsh, A.M., "A Course in Program Verification for Programmers," SIGCSE Bulletin, Volume 10, Number 1, February, 1978, pp. 213-216.

Kammann, R., "The Comprehensibility of Printed Instructions and Flowchart Alternative," Human Factors, Volume 17,2, 1975, pp. 183-191.

Keppel, G., and Saufly, W.H., Jr., Introduction to Design and Analysis, A Student's Handbook, San Francisco: W.H. Freeman and Co., 1980.

Knuth, D.E., "An Empirical Study of FORTRAN Programs," Software--Practice & Experience, Volume 1, Number 2, 1971, pp. 105-133.

Koltun, P., "DPL User's Manual," Technical Report TR82-004, Department of Computer Science, University of North Carolina, Chapel Hill, 1982.

________., "Introduction to Programming Using DPL," Technical Report TR82-005, Department of Computer Science, University of North Carolina, Chapel Hill, 1982.

Kreitzberg, C.B., and Swanson, L., "A Cognitive Model for Structuring an Introductory Programming Curriculum," AFIPS Conference Proceedings, Volume 43: 1974 National Computer Conference, Montvale, NJ: AFIPS Press, 1974, pp. 307-311.

Lambeth, D., "Use of Trace Specifications in the DPL Compiler," Master's thesis, Technical Report TR79-019, Department of Computer Science, University of North Carolina, Chapel Hill, 1979.

Larkin,J., McDermott, J., Simon, D.P., and Simon, H.A., "Expert and Novice Performance in Solving Physics Problems," Science, Volume 208, Number 20, June, 1980, pp. 1335-208.

Ledgard, H.F., Whiteside, J.A., Seymour, W., and Singer, A., "An Experiment on Human Engineering of Interactive Software," IEEE Transactions on Software Engineering, Volume SE-6, Number 6, 1980, pp. 602-604.

Lemos, R.S., "A Comparative Study of the Effectiveness of Team Interaction in COBOL Programming Language Learning," (Ph.D Dissertation, UCLA, 1977), Dissertation Abstracts International, Volume 38, 1977, pp. 2269B-2270B.

________., "Teaching Programming Languages: A Survey of Approaches," SIGCSE Bulletin, Volume 11, Number 1, February, 1979, pp. 174-181.

________., "An Implementation of Structured Walk-Throughs in Teaching COBOL Programming," Communications of the ACM, Volume 22, Number 6, 1979, pp. 335-340.

________., "Structured Walk-Throughs and Student Ratings of Faculty Effectiveness Versus Expediency," Journal of Educational Data Processing, Volume 16, Number 1, 1979, pp. 1-8.

________., "Measuring Programming Language Proficiency," AEDS Journal, Summer, 1980, pp. 261-273.

Linger, R.C., Mills, H.D., and Witt, B.I., Structured Programming, Reading, Mass.: Addison-Wesley, 1979.

Litecky, C., and Davis, G.B., "A Study of Errors, Error Proneness and Error Diagnosis in COBOL," Communications of the ACM, Volume 19, Number 1, 1976, pp. 33-37.

London, K.B., The People Side of Systems, London: McGraw-Hill, 1976.

Love, L.T., "Relating Individual Differences in Computer Programming Performance to Human Information Processing Abilities," (Ph.D Dissertation, University of Washington, 1977), Dissertation Abstracts International, Volume 38, 1977, p. 1443B

Lucas, L.C., and Kaplan, R.B., "A Structured Programming Experiment," Computer Journal, Volume 19, Number 2, 1976, pp. 136-138.

Madigan, S.A., "Intraserial Repetition and Coding Processes in Free Recall," Journal of Verbal Learning and Verbal Behavior, Volume 8, pp. 828-835, 1969.

Maurer, W.D., "The Teaching of Program Correctness," SIGCSE Bulletin, Volume 9, Number 1, February, 1977, pp. 142-144.

Mayer, R.E., "The Psychology of How Novices Learn Computer Programming," Computing Surveys, Volume 13, Number 1, 1981, pp. 121-141.

McCabe, T.J., "A Complexity Measure," IEEE Transactions on Software Engineering, Volume SE-2, Number 4, 1976, pp. 308-320.

Meissner, L.P., and Hinkins, R.L., "B4TRAN: A Structured Mini-Language Approach to the Teaching of FORTRAN," SIGCSE Bulletin, Volume 7, Number 1, February, 1975, pp. 200-205.

Miller, G.A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," Psychological Review, Volume 63, 1956, pp. 81-97.

Miller, L.A., Behavioral Studies of the Programming Process, National Technical Information Service Report #AD/A-061-633, October, 1978.

Mills, H.D., "Top Down Programming in Large Systems," Debugging Techniques in Large Systems, B. Rustin (Ed.), Englewood Cliffs: Prentice-Hall, 1971.

________., Mathematical Foundations for Structured Programming, IBM Report FSC 72-6013, 1972.

________., "How to Write Correct Programs and Know It," SIGPLAN Notices, Volume 10, Number 6, June, 1975, pp. 363-370.

________., _Computing Reviews_, Volume 17, Number 11, November, 1976, pp. 416-418.

T. Moher and G.M. Schneider, "Methods for Improving Experimentation in Software Engineering," _Sixth International Conference on Software Engineering_, New York: IEEE Press, 1981, pp. 224-233.

________., "Methodology and Experimental Results in Software Engineering," _International Journal of Man-Machine Studies_, Volume 16, Number 1, 1982, pp. 65-87.

Myers, G.J., "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," _Communications of the ACM_, Volume 21, Number 9, 1978, pp. 760-768.

Nagy, G., and Pennebaker, M.C., "Automatic Analysis of Student Programming Errors," _International Journal of Man-Machine Studies_, Volume 6, 1974, pp. 563-578.

Nanney, T.R., "Computer Science: An Essential Course for the Liberal Arts," _SIGCSE Bulletin_, Volume 8, Number 3, September, 1976, pp. 102-105.

Naur, P., "Proof of Algorithms by General Snapshots," _BIT_, Volume 6,4, 1966, pp. 310-316.

Newell, A., and Simon, H.A., _Human Problem Solving_, Englewood Cliffs: Prentice-Hall, 1972.

Newsted, P.R., "Grade and Ability Predictions in an Introductory Programming Course," _SIGCSE Bulletin_, Volume 7, Number 2, June, 1975, pp. 87-91.

________., "FORTRAN Program Comprehension as a Function of Documentation," School of Business Administration, University of Wisconsin, Milwaukee, undated.

Newton, G.E., and Starkey, J.D., "Teaching Both PL/1 and FORTRAN to Beginners," _SIGCSE Bulletin_, Volume 8, Number 3, September, 1976, pp. 106-107.

Parnas, D.L., "Dubiety of Increased Funding for Experimental Computer Science," _Communications of the ACM_, Volume 24, Number 3, 1981, pp. 162-163.

Perlis, A.J., _Introduction to Computer Science_, New York: Harper & Row, 1975.

Petersen, C.G., and Howe, T.G., "Predicting Academic Success in Introduction to Computers," _AEDS Journal_, Fall, 1979, pp. 182-191.

Proceedings of the International Conference on Reliable Software, _SIGPLAN Notices_, Volume 10, Number 6, June, 1975.

Ramsey, H.R., Atwood, M.E., and Van Doren, J.R., _A Comparative Study of Flowcharts and Program Design Languages for the Detailed Procedural Specification of Computer Programs_ Denver: Science Applications, Inc., 1978.

Reisner, P., "Use of Psychological Experimentation as an Aid to Development of a Query Language," _IEEE Transactions on Software Engineering_, Volume SE-3, Number 3, 1977, pp. 218-229.

Sackman, H., Erikson, W.J., and Grant, E.F., "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," _Communications of the ACM_, Volume 11, Number 1, 1968, pp. 3-11.

Sackman, H., _Man-Computer Problem Solving_, Princeton: Auerbach Publishers, 1970.

Schneider, G.M., "The Introductory Programming Course in Computer Science -- Ten Principles," _SIGCSE Bulletin_, Volume 10, Number 1, February, 1978, pp. 107-114.

Schneider, G.M., Sedlmeyer, R.L., and Kearney, J., "On the Complexity of Measuring Software Complexity," _National Computer Conference Proceedings, 1981_, Arlington, Va.: AFIPS Press, 1981, pp. 317-322.

Sheil, B.A., "The Psychological Study of Programming," _Computing Surveys_, Volume 13, Number 1, 1981, pp. 101-120.

Sheppard, S.B., Curtis, B., Milliman, P., and Love, T., "Modern Coding Practices and Programmer Performance," _Computer_, Volume 12, Number 12, 1979, pp. 41-49.

Sheppard, S.B., Kruesi, E., and Curtis, B., "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications," _Proceedings of the Sixth International Conference on Software Engineering_, (New York: IEEE Press, 1981), pp. 207-214.

Shneiderman, B., "Exploratory Experiments in Programmer Behavior," _International Journal of Computer and Information Science_, Volume 5, Number 2, 1976, pp. 123-143.

________., "Measuring Computer Program Quality and Comprehension," _International Journal of Man-Machine Studies_, Volume 9, 1977, pp. 465-478.

Shneiderman, B., Mayer, R., McKay, D., and Heller, P., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," Communications of the ACM, Volume 20,6, 1977, pp. 373-381.

Shneiderman, B., Software Psychology, Cambridge: Winthrop Publishers, 1980.

SIGSOFT First Software Engineering Symposium on Tool and Methodology Evaluation, "Proposals for Tool and Methodology Evaluation Experiments," Software Engineering Notes, Volume 7, Number 1, 1982, pp. 6-75.

Sime, M.E., Green, T.R.G., and Guest, D.J., "Scope Marking in Computer Conditionals -- A Psychological Evaluation," International Journal of Man-Machine Studies, Volume 9, Number 1, 1977, pp. 107-118.

Simon, H.A., "How Big Is a Chunk?" Science, Volume 183, 1974, pp. 482-488.

Skelton, J.E., "Time-Sharing Versus Batch Processing and Teaching Beginning Computer Programming: An Experiment," AEDS Journal, March, 1972, pp.91-97, and June, 1972, pp. 103-109.

Smith, L.B., "A Comparison of Batch Processing and Instant Turnaround," Communications of the ACM, Volume 10, Number 8, 1967, pp. 495-500.

Stoddard, S.D., Sedlmeyer, R.L., and Lee, R.G., "Breadth or Depth in Introductory Computer Courses: A Controlled Experiment." SIGCSE Bulletin, Volume 11, Number 1, February, 1979, pp. 41-44.

Turski, W.M., (ed.), Programming Teaching Techniques, New York: American Elsevier Publishing Co., 1973.

Ulloa, M., "Teaching and Learning Computer Programming: A Survey of Student Problems, Teaching Methods, and Automated Instructional Tools," SIGCSE Bulletin, Volume 12, Number 2, July, 1980, pp. 48-64

Walsh, T.J., "A Software Reliability Study Using a Complexity Measure," AFIPS Conference Proceedings, Volume 48: 1979 National Computer Conference, Montvale, NJ: AFIPS Press, 1979, pp. 761-769.

Weinberg, G., The Psychology of Computer Programming, New York: Van Nostrand Reinhold, Co., 1971.

________., "The Psychology of Improved Programming Performance," Datamation, November, 1972, pp. 82-85.

Weiner, L.H., "The Roots of Structured Programming," SIGCSE Bulletin, Volume 10, Number 1, February, 1978, pp. 243-254.

Weiss, D.M., "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," The Journal of Systems and Software, Volume 1, 1979, pp. 57-70.

Wulf, W.A., Shaw, M., Hilfinger, P., and Flon, L., Fundamental Structures of Computer Science, Reading, Mass.: Addison-Wesley, 1981.

Youngs, E.A., "Human Errors in Programming," International Journal of Man-Machine Studies, Volume 6,4, 1974, pp. 361-376.

Zolnowski, J.C., and Simmons, D.B., "Taking the Measure of Program Complexity," National Computer Conference Proceedings, 1981, Arlington, Va.: AFIPS Press, 1981, pp. 329-336.