ED 112 861 IR 002 570

AUTHOR	Friend, Jamesine
TITLE	Programs Students Write. Technical Report No. 257.
INSTITUTION	Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY	Advanced Research Projects Agency (DOD), Washington, D.C.; Office of Naval Research, Washington, D.C. Personnel and Training Research Programs Office.
REPORT NO	SU-IMSSS-TR-257
PUB DATE	25 Jul 75
NOTE	281p.; Psychology and Education Series
EDRS PRICE	MF-$0.76 HC-$14.59 Plus Postage
DESCRIPTORS	College Students; *Computer Assisted Instruction; *Computer Programs; *Computer Science Education; Educational Assessment; Educational Programs; Educational Technology; Higher Education; *Problem Solving; Programing; Statistical Data; *Student Developed Materials; Teaching Techniques; *Tutorial Programs

ABSTRACT

To explore the problem of designing an automated system for instruction in programing, and to study the problem-solving behavior of students, computer programs written by 40 college students as part of a CAI course in Algebraic Interpretive Dialogue were analyzed. The self-contained course consisted of 50 tutorial lessons; the analysis covers programs written as solutions to 25 programing problems, including 747 problems containing 7,063 commands. The distribution of data over problems and over students is discussed, along with problem difficulty and diversity of student solutions. (Author/SK)

IR

PROGRAMS STUDENTS WRITE

BY

JAMESINE FRIEND

TECHNICAL REPORT NO. 257

JULY 25, 1975

PSYCHOLOGY AND EDUCATION SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

2008 570

TECHNICAL REPORTS

PSYCHOLOGY SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

(Place of publication shown in parentheses; if published title is different from title of Technical Report, this is also shown in parentheses.)

165 L. J. Hubert. A formal model for the perceptual processing of geometric configurations. February 19, 1971. (A statistical method for investigating the perceptual confusions among geometric configurations. Journal of Mathematical Psychology, 1972, 9, 389-403.)

166 J. F. Juola, I. S. Fischler, C. T. Wood, and R. C. Atkinson. Recognition time for information stored in long-term memory. (Perception and Psychophysics, 1971, 10, 8-14.)

167 R. L. Klatzky and R. C. Atkinson. Specialization of the cerebral hemispheres in scanning for information in short-term memory. (Perception and Psychophysics, 1971, 10, 335-338.)

168 J. D. Fletcher and R. C. Atkinson. An evaluation of the Stanford CAI program in initial reading (grades K through 3). March 12, 1971. (Evaluation of the Stanford CAI program in initial reading. Journal of Educational Psychology, 1972, 63, 597-602.)

169 J. F. Juola and R. C. Atkinson. Memory scanning for words versus categories. (Journal of Verbal Learning and Verbal Behavior, 1971, 10, 522-527.)

170 I. S. Fischler and J. F. Juola. Effects of repeated tests on recognition time for information in long-term memory. (Journal of Experimental Psychology, 1971, 91, 54-58.)

171 P. Suppes. Semantics of context-free fragments of natural languages. March 30, 1971. (In K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes (Eds.), Approaches to natural language. Dordrecht: Reidel, 1973. Pp. 221-242.)

172 J. Friend. INSTRUCT coders' manual. May 1, 1971.

173 R. C. Atkinson and R. M. Shiffrin. The control processes of short-term memory. April 19, 1971. (The control of short-term memory. Scientific American, 1971, 224, 82-90.)

174 P. Suppes. Computer-assisted instruction at Stanford. May 19, 1971. (In Man and computer. Proceedings of international conference, Bordeaux, 1970. Basel: Karger, 1972. Pp. 298-330.)

175 D. Jamison, J. D. Fletcher, P. Suppes, and R. C. Atkinson. Cost and performance of computer-assisted instruction for education of disadvantaged children. July, 1971.

176 J. Offir. Some mathematical models of individual differences in learning and performance. June 28, 1971. (Stochastic learning models with distribution of parameters. Journal of Mathematical Psychology, 1972, 9(4),)

177 R. C. Atkinson and J. F. Juola. Factors influencing speed and accuracy of word recognition. August 12, 1971. (In S. Kornblum (Ed.), Attention and performance IV. New York: Academic Press, 1973.)

178 P. Suppes, A. Goldberg, G. Kanz, B. Searle, and C. Stauffer. Teacher's handbook for CAI courses. September 1, 1971.

179 A. Goldberg. A generalized instructional system for elementary mathematical logic. October 11, 1971.

180 M. Jerman. Instruction in problem solving and an analysis of structural variables that contribute to problem-solving difficulty. November 12, 1971. (Individualized instruction in problem solving in elementary mathematics. Journal for Research in Mathematics Education, 1973, 4, 6-19.)

181 P. Suppes. On the grammar and model-theoretic semantics of children's noun phrases. November 29, 1971.

182 G. Kreisel. Five notes on the application of proof theory to computer science. December 10, 1971.

183 J. M. Moloney. An investigation of college student performance on a logic curriculum in a computer-assisted instruction setting. January 28, 1972.

184 J. E. Friend, J. D. Fletcher, and R. C. Atkinson. Student performance in computer-assisted instruction in programming. May 10, 1972.

185 R. L. Smith, Jr. The syntax and semantics of ERICA. June 14, 1972.

186 A. Goldberg and P. Suppes. A computer-assisted instruction program for exercises on finding axioms. June 23, 1972. (Educational Studies in Mathematics, 1972, 4, 429-449.)

187 R. C. Atkinson. Ingredients for a theory of instruction. June 26, 1972. (American Psychologist, 1972, 27, 921-931.)

188 J. D. Bonvillian and V. R. Charrow. Psycholinguistic implications of deafness: A review. July 14, 1972.

189 P. Arabie and S. A. Boorman. Multidimensional scaling of measures of distance between partitions. July 26, 1972. (Journal of Mathematical Psychology, 1973, 10,)

190 J. Ball and D. Jamison. Computer-assisted instruction for dispersed populations: System cost models. September 15, 1972. (Instructional Science, 1973, 1, 469-501.)

191 W. R. Sanders and J. R. Ball. Logic documentation standard for the Institute for Mathematical Studies in the Social Sciences. October 4, 1972.

192 M. T. Kane. Variability in the proof behavior of college students in a CAI course in logic as a function of problem characteristics. October 6, 1972.

193 P. Suppes. Facts and fantasies of education. October 18, 1972. (In M. C. Wittrock (Ed.), Changing education: Alternatives from educational research. Englewood Cliffs, N. J.: Prentice-Hall, 1973. Pp. 6-45.)

194 R. C. Atkinson and J. F. Juola. Search and decision processes in recognition memory. October 27, 1972.

195 P. Suppes, R. Smith, and M. Léveillé. The French syntax and semantics of PHILIPPE, part 1: Noun phrases. November 3, 1972.

196 D. Jamison, P. Suppes, and S. Wells. The effectiveness of alternative instructional methods: A survey. November, 1972.

197 P. Suppes. A survey of cognition in handicapped children. December 29, 1972.

198 B. Searle, P. Lorton, Jr., A. Goldberg, P. Suppes, N. Ledet, and C. Jones. Computer-assisted instruction program: Tennessee State University. February 14, 1973.

199 D. R. Levine. Computer-based analytic grading for German grammar instruction. March 16, 1973.

200 P. Suppes, J. D. Fletcher, M. Zanotti, P. V. Lorton, Jr., and B. W. Searle. Evaluation of computer-assisted instruction in elementary mathematics for hearing-impaired students. March 17, 1973.

201 G. A. Huff. Geometry and formal linguistics. April 27, 1973.

202 C. Jensema. Useful techniques for applying latent trait mental-test theory. May 9, 1973.

203 A. Goldberg. Computer-assisted instruction: The application of theorem-proving to adaptive response analysis. May 25, 1973.

204 R. C. Atkinson, D. J. Herrmann, and K. T. Wescourt. Search processes in recognition memory. June 8, 1973.

205 J. Van Campen. A computer-based introduction to the morphology of Old Church Slavonic. June 18, 1973.

206 R. B. Kimball. Self-optimizing computer-assisted tutoring: Theory and practice. June 25, 1973.

207 R. C. Atkinson, J. D. Fletcher, E. J. Lindsay, J. O. Campbell, and A. Barr. Computer-assisted instruction in initial reading. July 9, 1973.

208 V. R. Charrow and J. D. Fletcher. English as the second language of deaf students. July 20, 1973.

209 J. A. Paulson. An evaluation of instructional strategies in a simple learning situation. July 30, 1973.

210 N. Martin. Convergence properties of a class of probabilistic adaptive schemes called sequential reproductive plans. July 31, 1973.

PROGRAMS STUDENTS WRITE

by

Jamesine Friend

This research was supported jointly by:

Office of Naval Research
Psychological Sciences Division
Personnel and Training Research Programs (Code 458)
Contract Authority Number: NR 154-326
Scientific Officers: Dr. Marshall Farr and Dr. Joseph Young

and

Advanced Research Projects Agency
ARPA Order Number: 2284 dated 30 August 1972
Program Code Number: 3D20

Contract Number:

N00014-67-A-0012-0054
1 August 1972 - 31 July 1975

Principal Investigator:

Richard C. Atkinson
Professor of Psychology
Institute for Mathematical Studies in the Social Sciences
Stanford University
Stanford, California 94305
(415) 497-4117

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report 257	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Programs Students Write		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jamesine Friend		8. CONTRACT OR GRANT NUMBER(s) N00014-67-A-0012-0054
9. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Mathematical Studies in the Social Sciences - Stanford University Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N RR 042-0; RR 042-0-0 NR 154-326
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel & Training Research Programs Office of Naval Research (Code 458) Arlington, VA 22217		12. REPORT DATE July 25, 1975
		13. NUMBER OF PAGES 268
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer-assisted instruction, problem solving, problem difficulty, program diversity, Algebraic Interpretive Dialogue, instruction in programming, error analysis, solution analysis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The present study addresses itself to the problem of designing an automated system for instruction in programming, and also to the study of problem-solving behavior, as exhibited by students using a CAI course in computer programming. The study uses computer programs written by 40 college students during the winter and spring quarters of 1972 as part of a CAI course in AID (Algebraic Interpretive Dialogue), an algebraic language similar to BASIC. The course is self-contained and consists of 50 tutorial lessons described in detail in Friend (1973). The programs analyzed were written as solutions to 25 programming problems from the course; 747 solutions containing 7063 commands were analyzed. The distribution of the data over problems and over students is discussed. Problem difficulty and diversity of student solutions are also discussed in detail.

SUMMARY

The present study addresses itself to the problem of designing an automated system for instruction in programming, and also to the study of problem-solving behavior, as exhibited by students using a CAI course in computer programming.

The study uses computer programs written by 40 college students during the winter and spring quarters of 1972 as part of a CAI course in AID (Algebraic Interpretive Dialogue), an algebraic language similar to BASIC. The course is self-contained and consists of 50 tutorial lessons described in detail in Friend (1973).

The programs analyzed were written as solutions to 25 programming problems from the course; 747 solutions containing 7063 commands were analyzed. The distribution of the data over problems and over students is discussed. Problem difficulty and diversity of student solutions are also discussed in detail.

CHAPTER I

Introduction

One of the major design problems in implementing computer-assisted instruction (CAI) courses in computer programming is that of analyzing student-written programs in real time. An instructional program capable of providing response-sensitive, specific corrective instruction for student programming efforts should have several attributes, including the ability to identify overt errors and to deliver unambiguous error messages, the ability to determine whether or not a student's program is a correct solution, and the ability to determine from a partially written or nonfunctioning program the strategy preferred by the student and to give assistance on that basis.

The first of these attributes--the ability to identify overt errors and to deliver the appropriate messages--is usually seen as one of the functions of the compiler or interpreter. Syntax errors and a few kinds of semantic errors fall in this category. Overt semantic errors that are problem-dependent (such as the use of an incorrect algebraic formula) cannot be relegated to the interpreter; they must be handled by a routine that has access to curriculum data. However, no 'perfect' algorithm for the detection of problem-dependent errors can be written, and an 'ideal programming consultant' of the type described above cannot be unambiguously defined.

Similarly, an algorithm that provides the second 'ideal' attribute, the ability to determine the correctness of a student's solution, does not exist. It may be possible, however, to design a strategy that usefully approaches this goal, for practical purposes, and it is one of the aims of this study to investigate how close such a heuristic solution might come.

As for the third desired attribute of an ideal programming consultant--the problem of guiding a student along his chosen path--the problem itself is poorly defined and the final chapters of this report (in which a study of the diversity of correct solutions is presented) attempt to provide a greater understanding of the problem and to present a research method and results ~~that~~ may lead to a workable solution.

The present study addresses itself to the problem of designing an automated system for instruction ~~in~~ programming, and also to the study of problem-solving behavior, as exhibited by students using a CAI course in computer programming.

The study uses computer programs written by 40 college students during the winter and spring quarters of 1972 as part of a CAI course in AID (Algebraic Interpretive Dialogue), an algebraic language similar to BASIC. The course is self-contained and consists of 50 tutorial lessons described in detail in Friend (1973a).

The AID course consists of two computer programs, one that presents instructional material to the student, and a second, the AID interpreter, which the student uses when writing and debugging his own AID programs. The latter program was provided by Digital Equipment Corporation, manufacturer of the computer system on which the instructional system is implemented. The interpreter was modified to allow for the collection of student responses.

The programs analyzed were written as solutions to 25 programming problems from the AID course. They were chosen to test programming ability and were expected to be among the most difficult problems in the course--an expectation that was confirmed. Not all students attempted every problem; in all, 747 solutions containing 7063 programming commands were analyzed.

The format and method of presentation are the same for all of the problems. After an introductory instruction, a problem is stated in simple English by the instructional program. The student types the command 'AID' to call the AID interpreter, and he then attempts to write and debug a program to solve the given problem. During the time he uses the AID interpreter, he is interacting with the computer as a professional programmer would. His attempts are not monitored by the instructional program, and he is free to use any programming devices he chooses (he may even write programs that are unrelated to the given problem). The only instruction he receives while using the AID interpreter is in the form of error messages given by the interpreter if he attempts to execute an incorrectly formed command or program. The student may write and execute any syntactically correct AID commands and may delete or replace any commands. He may also file programs in disk storage or recall previously filed programs. When he has completed the program to his satisfaction--or has given up--he recalls the instructional program by typing the command 'INST'.

The instructional program, which does not have access to the student's program nor the ability to analyze it, attempts to determine whether the solution is correct by asking for selected results obtained during execution of the program. If the student's report of the performance of his program indicates that the solution is correct, the lessons continue. If not, additional instruction is given and the student may be asked to call the AID interpreter again for another trial.

The sequence of instruction is under the control of the student. He may work exercises in any sequence, skipping some exercises, returning to others for review, etc. Hence, a student may encounter a problem several times.

The amount of instruction varies, depending both upon student performance and upon the desires of the student. Usually, after an incorrect response to an item one short corrective message is given automatically. The student may call for additional instruction, in the form of 'hints', by typing a question mark. Some of the hints contain explicit help and occasionally even give a complete correct solution to the problem. Because of the variation of the amount of instruction intervening between the student's first trial at a given programming problem and subsequent trials only the data collected during first trial have been analyzed in detail.

Hints are also available to the student before his first attempt and again, some of the hints are quite explicit. (The use of hints is discussed, with examples, in Chapter II.) For this reason, and because students are free to chart their own pathway through the course, the amount of instruction received before the first trial varies and affects the proportion of correct solutions, a fact that must be kept in mind when interpreting the results reported here.

An illustration of a student's interaction with the instructional program and the AID interpreter follows (the problem is real, the student hypothetical). In this example, and throughout this report, the problem and the student's responses are shown in upper case as they appear on the Model-33 Teletype, the terminal most commonly used for the AID course. The asterisk to the left of the student's response signifies to the student that the system is ready to receive input. The problem shown below is taken from Lesson 11 and is one of the problems used in the research reported here.

```
WRITE A PROGRAM THAT WILL LIST THE
RADIUS, DIAMETER, CIRCUMFERENCE, AND
AREA OF A CIRCLE OF RADIUS R. THEN
USE THE PROGRAM FOR R = 10, 20, 30,
40, AND 50.
```

The instructional program states the problem.

```
* ?
```

The student immediately asks for a hint...

```
USE THESE FORMULAS (R STANDS FOR
RADIUS):
    D = 2 * R
    C = 2 * 3.14159265 * R
    A = 3.14159265 * R↑2
```

...which is given by the instructional program. Formulas known by most students are given in the hints rather than in the problem statements.

```
*AID
```

The student then calls the AID interpreter.

```
*SET D = 2*R
```

The student starts to write a program but inadvertently omits the step number...

```
R = ?
```

...and receives an error message indicating that he attempted to use an undefined variable.

```
*5.1 SET D = 2*R
*5.2 SET C = 2*3.14*R
*5.3 SET A = 3.14*D↑2
*5.4 TYPE R, D, C, A
```

The student corrects the error and continues to write the program without syntax errors.

```
*DO PART 5
```

The student tries to execute the program, but fails to specify a value for R....

```
ERROR AT STEP 5.1: R = ?
```

...and receives an error message.

```
*DO PART 5 FOR R = 1
   R = 1
   D = 2
   C = 6.28
   A = 12.56
```

The student correctly executes the program.

```
*5.3 SET A = 3.14*R↑2
```

The student replaces Step 5.3 to correct an algebraic error...

```
*DO PART 5 FOR R = 10(10)50
   R = 10
   D = 20
   C = 62.8
   A = 314
   R = 20
   .
   .
   .
```

...and then executes the program for the values of R specified in the problem.

*INST	After the program stops, the student calls the instructional program...
WHAT IS THE CIRCUMFERENCE OF A CIRCLE WITH RADIUS 30?	...which asks for a selected result.
*188.4	The student reports the result he obtained...
CORRECT	...and is judged correct by the instructional program. The instructional program used 3.14159265 as an approximation for π rather than 3.14 as used by the student. For this problem, any response that agrees with the coded correct answer to four significant digits is considered correct.

Had the student's response to the last question indicated that his program was not functioning correctly, he would have been told to ask for as many hints as he needed and then to try again. The algebraic formulas would have been repeated in the hints, with the final hint giving a correct solution that the student could copy.

Had the above work been done by a real student, all student input between the two commands 'AID' and 'INST' would have been stored as data by the AID interpreter. (Other responses, including the initial request for a hint, would have been collected by the instructional program, and are not analyzed in this report.) Except for characters or lines erased by the student immediately after they are typed, every character typed by the student is recorded. In addition, a small amount of bookkeeping information--the student's identification, the problem number, and the date and time--is stored with each data block.

The data are presented in some detail in Chapter III, following the description of the programming problems in Chapter II. The distribution of the data over problems and over students is discussed, summary statistics of the number of AID commands typed and the number of commands executed are given, and the number of occurrences of different kinds of AID commands is reported and compared with the predicted proportions of the kinds of commands.

In the remaining chapters, two different kinds of analysis are presented. Discussion of problem difficulty occupies Chapters IV, V, and VI. Chapter IV describes several methods for determining the proportion of correct and partially correct solutions, and derives the statistics that are used later in the development of formulas for measuring problem difficulty; the distribution of correct solutions is also discussed. Chapter V reports an analysis of overt errors. Errors are classified as either syntactic or semantic, and each of these classes is further subdivided. The distribution of errors over students is shown, and a measure of error rates developed. Both the number of errors and the error rates are used in measuring problem difficulty. In Chapter VI, 19 measures of problem difficulty are defined, and the correlations between pairs of measures are given. Ten characteristics of the problem that might affect problem difficulty are discussed and measured. These ten characteristics were used as independent variables in stepwise multiple linear regressions from which linear formulas for predicting problem difficulty were developed. The analysis of problem difficulty reported is similar to that in Moloney (1972) of proofs written by students in a CAI course in logic.

The second kind of analysis is a study of the diversity of student solutions, and in this I am indebted to Dr. Michael Kane (1972) for the methods that he developed for a similar analysis of proofs produced by students in the CAI logic course. The use of equivalence relations reported here is adapted from Kane but the method of measuring diversity is somewhat different from Kane's method of measuring variability of student-written proofs.

Equivalence of programs is discussed in Chapter VII. Four definitions of equivalence are given, and the 551 correct and nearly correct solutions found in the data are classified by each of the four definitions. In Chapter VIII, four measures of diversity of solutions are defined, and the effect on diversity of several characteristics of the problems and the curriculum is investigated. Stepwise multiple linear regressions were then used to develop linear formulas for the prediction of diversity. The final chapter summarized the findings of this study and discusses their implications.

CHAPTER II

Description of Programming Problems

The 25 programming problems used for this study are displayed below, together with expected correct solutions. The solutions given are those anticipated as the most likely correct solution to each problem. (The chapter closes with a discussion of why these solutions were chosen.) The programs written by students (Appendix A) are discussed in Chapters VII and VIII.

Although comments on the context in which each problem appeared are included, there are ~~no~~ explanations of the AID programs. A description of the language is given in Friend (1973). Each problem is identified by the lesson number and the problem used in the course; the identifier L 16-4, for example, refers to Lesson 16, Problem 4. The optional hints are shown following the problem statement. A student who asked for a hint received the first hint listed. A second request brought the second hint, and so on until the hints were exhausted.

L 5-30: 1 CENTIMETER = .3937 INCHES. CONVERT THE FOLLOWING LENGTHS TO CENTIMETERS:

```
     6.9 INCHES
     7.445 INCHES
     23.9753 INCHES
```

Hint #1:
X CENTIMETERS = Y INCHES/.3937

Hint #2:
TO CONVERT 5 INCHES TO CENTIMETERS, DIVIDE 5 BY .3937.

Correct solution:

```
     SET K = .3937
     TYPE 6.9/K, 7.445/K, 23.9753/K
```

Technically, Problem L 5-30 is not a programming problem, because the concept of a stored program is introduced later, and the student is expected to use direct (immediately executed) commands to solve the problem. Only two AID commands, TYPE and SET, have been introduced by Lesson 5, and those only in their direct form. When the student's solutions were graded for this study, a SET command was not required, since this was not requested in the problem statement.

L 8-9: USE A "LET" COMMAND TO DEFINE A FUNCTION THAT GIVES THE RECIPROCAL OF X. USE YOUR FUNCTION TO FIND THE RECIPROCAL OF

```
     119.4
     67.3↑3
     6 + 4
```

Hint #1:
THE RECIPROCAL OF X IS 1/X.

Hint #2:
USE THE COMMAND

```
     LET R(X) = 1/X
```

TO DEFINE THE FUNCTION.

Hint #3:
USE THESE COMMANDS:

```
     LET R(X) = 1/X
     TYPE R(119.4)
     TYPE R(67.3↑3)
     TYPE R(6 + 4)
```

Correct solution:

```
     LET F(X) = 1/X
     TYPE F(119.4)
     TYPE F(67.3↑3)
     TYPE F(6 + 4)
```

This problem is more nearly a 'programming problem' than the preceding one, since it requires the use of a stored formula as a specification of an algorithm. The use of the LET command was introduced in Lesson 8, and this is the second problem in which students were required to use such a command. Unless a student defined and used a function for this problem, his solution was considered incorrect since the problem statement specified that a LET command was to be used.

L 8-27: DEFINE A "VOLUME" FUNCTION THAT WILL GIVE THE VOLUME OF A CYLINDRICAL TANK OF RADIUS R AND HEIGHT H. (VOLUME = 3.1416 TIMES

THE RADIUS SQUARED TIMES THE HEIGHT.)
FIND THE VOLUME OF 2 TANKS:
    TANK A IS 57.5 FEET HIGH AND HAS A RADIUS OF 18.6 FEET.
    TANK B IS 65.4 FEET HIGH AND THE RADIUS IS 19.3 FEET.

Hint #1:
USE THIS COMMAND TO DEFINE THE "VOLUME" FUNCTION:
```
    LET V(R, H) = 3.1416*R↑2*H
```

Hint #2:
AFTER THE VOLUME FUNCTION IS DEFINED, USE THIS COMMAND TO FIND
THE VOLUME OF TANK A:
```
    TYPE V(18.6, 57.5)
```

Correct solution:
```
    LET V(R,H) = 3.1416*R↑2*H
    TYPE V(18.6,57.5), V(19.3,65.4)
```

This is the first problem in which a function of two variables is used. Note that the formula for the volume of a right cylinder is given in the problem statement; formulas that are likely to be known by most students (e.g., area of a circle) are not ordinarily given in the problem, but are given in the optional hints.

L 8-28: DEFINE A FUNCTION TO CONVERT DEGREES FAHRENHEIT TO DEGREES CENTIGRADE. THEN CONVERT THESE TEMPERATURES TO CENTIGRADE:
    0, 10, 32, 72, 212

Hint #1:
TO CONVERT TO CENTIGRADE SUBTRACT 32 AND MULTIPLY BY 5/9.

Hint #2:
DEFINE A "CONVERSION TO CENTIGRADE" FUNCTION LIKE THIS:
```
    LET C(F) = 5/9*(F - 32)
```
WHERE F STANDS FOR "DEGREES FAHRENHEIT."

Correct solution:
```
    LET C(F) = 5/9*(F - 32)
    TYPE C(0), C(10), C(32), C(72), C(212)
```

This problem is similar to L 8-9, which also required the use of a function of one variable, although here the formula is somewhat more complex.

L 9-3: WRITE A FUNCTION H(A, B) THAT WILL FIND THE LENGTH OF THE HYPOTENUSE OF A RIGHT TRIANGLE IF THE LENGTHS OF THE OTHER TWO SIDES ARE GIVEN BY A AND B. TEST YOUR FUNCTION ON THESE TRIANGLES:

1. A = 3, B = 4
2. A = 12, B = 12
3. A = 1/2, B = 3/4
4. A = 9, B = 13.2

Hint #1:
THE HYPOTENUSE OF A RIGHT TRIANGLE IS EQUAL TO THE SQUARE ROOT OF THE SUM OF THE SQUARES OF THE OTHER TWO SIDES.

Hint #2:
HYPOTENUSE = SQUARE ROOT((SIDE A)↑2 + (SIDE B)↑2)

Correct solution:
```
     LET H(A, B) = SQRT(A↑2 + B↑2)
     TYPE H(3, 4), H(12, 12), H(1/2, 3/4), H(9, 13.2)
```

The standard functions SGN, SQRT, IP, and FP were introduced in Lesson 9; this is the first programming problem that uses SQRT, and also the first problem that allows the student to define a function in terms of a standard function. However, since the problem did not specify using the function SQRT, solutions that used other algebraic formulations were also considered correct.

L 9-8: WHEN AN INTEGER M IS DIVIDED BY AN INTEGER N, THERE IS A QUOTIENT AND A REMAINDER. WRITE A QUOTIENT FUNCTION Q(M, N). USE THE FUNCTION TO FIND THE QUOTIENTS FOR THESE VALUES OF M AND N:

M = 9172	N = 38
M = 13	N = 87
M = 768	N = 101
M = 6480	N = 15

Hint #1:
FOR EXAMPLE: 14/3 HAS A QUOTIENT OF 4 AND A REMAINDER OF 2.

Hint #2:
USE THE IP FUNCTION TO FIND THE QUOTIENT.

Correct solution:
```
     LET Q(M, N) = IP(M/N)
     TYPE Q(9172,38), Q(13,87), Q(768,101), Q(6480,15)
```

This is the first problem requiring the use of IP, the 'integer part' function.

L 10-12: WRITE AN INDIRECT STEP THAT WILL CONVERT MILES PER HOUR TO FEET PER SECOND. THEN CONVERT ALL OF THESE TO FEET PER SECOND:

```
     10 MILES PER HOUR
     100 MILES PER HOUR
     65 MILES PER HOUR
     1023 MILES PER HOUR
```

Hint:
IF S STANDS FOR SPEED IN MILES PER HOUR, THIS COMMAND WILL GIVE THE SPEED IN FEET PER SECOND.

```
     TYPE S*5280/(60*60)
```

Correct solution:

```
     3.1 TYPE S*5280/(60*60)
     DO STEP 3.1 FOR S = 10, 100, 65, 1023
```

With the introduction of the concept of stored commands and their execution in Lesson 10, the first true programming problem in the course is presented.

L 10-19: WRITE AN INDIRECT STEP THAT WILL TYPE THE SQUARE ROOT OF X. DO THE STEP FOR X = 1, 2, 3, ..., 10.

Hint:
IN THE SEQUENCE

```
     1, 2, 3, ..., 10
```

THE INITIAL VALUE IS 1
THE STEP SIZE IS 1
THE FINAL VALUE IS 10

Correct solution:

```
     5.4 TYPE SQRT(X)
     DO STEP 5.4 FOR X = 1(1)10
```

This problem is similar to the preceding one, which also requires that a single stored command be iterated several times. The instruction intervening between these two problems explains the use of the range specification in FOR modifiers, and it was anticipated that students would use that device in solving this problem.

L 11-11: WRITE A PROGRAM THAT WILL LIST THE RADIUS, DIAMETER, CIRCUMFERENCE, AND AREA OF A CIRCLE X OF RADIUS R. THEN USE THE PROGRAM FOR R = 10, 20, 30, 40, AND 50.

Hint:
USE THESE FORMULAS (R STANDS FOR RADIUS):

```
    D = 2*R                 (D = DIAMETER)
    C = 2*3.14159265*R      (C = CIRCUMFERENCE)
    A = 2.14159265*R↑2      (A = AREA)
```

Correct solution:

```
    2.1 SET D = 2*R
    2.2 SET C = 2*3.14159265*R
    2.3 SET A = 3.14159265*R↑2
    2.4 TYPE R, D, C, A
    DO PART 2 FOR R = 10(10)50
```

In Lesson 11, the student is taught how to write and execute sequences of stored commands. This is the first problem that requires the student to write a program consisting of more than a single command.

L 12-4: WRITE A PROGRAM THAT WILL ASK YOU FOR 3 NUMBERS, A, B, AND C, AND THEN GIVE YOU THE AVERAGE OF THE 3 NUMBERS. AFTER YOU HAVE TESTED YOUR PROGRAM USE IT TO FIND THE AVERAGE OF

```
    A = 179.053
    B = 23.7
    C = 271.0015
```

Hint:
TO FIND THE AVERAGE OF 3 NUMBERS, ADD THE 3 NUMBERS TOGETHER AND DIVIDE THE SUM BY 3.

Correct solution:

```
    2.1 DEMAND A
    2.2 DEMAND B
    2.3 DEMAND C
    2.4 TYPE (A + B + C)/3
    DO PART 2
```

The DEMAND command has just been introduced and this is the first problem that gives the student the opportunity to use it.

L 13-29: WRITE A PROGRAM THAT WILL CONVERT YEARS TO MONTHS. THE PROGRAM SHOULD ASK FOR THE NUMBER OF YEARS AND THEN PRINT THE NUMBER OF MONTHS.
USE YOUR PROGRAM TO FIND THE NUMBER OF MONTHS IN

```
    2 YEARS
    16 YEARS
    100 YEARS
    2593. YEARS
```

Correct solution:

```
    8.1 DEMAND Y
    8.2 TYPE Y*12
    DO PART 8, 4 TIMES
```

Lesson 13 is one of seven tests contained in the course. Because this problem is a test item, no hints are provided. Although the problem is similar to Problem L 12-4, notice that it was anticipated that students would use a TIMES modifier, which was introduced in Lesson 12 after Problem L 12-4.

L 15-15: WRITE A PROGRAM THAT WILL FIND THE SMALLER OF TWO NUMBERS X AND Y.

Hint:
REWRITE THE PROGRAM IN PROBLEM 14 SO THAT IT TYPES THE SMALLER NUMBER RATHER THAN THE LARGER.

Correct solution:

```
    2.1 DEMAND X
    2.2 DEMAND Y
    2.3 TYPE X IF X < Y
    2.4 TYPE Y IF X > Y
    DO PART 2
```

The problem immediately preceding this one serves as an example of a program that types the larger of two numbers. The example is identical to the correct solution given above except that the symbols > and < are interchanged. The conditional clause was introduced in Lesson 15 and, except for a problem that requires the student to copy a program given in its entirety in the text, this is the first problem that uses conditional commands. In grading the solutions to this problem, the student's program is not required to provide for the case X = Y.

L 15-17: WRITE A PROGRAM THAT FINDS THE SMALLEST OF 4 NUMBERS.

Hint:
CHANGE PART 51 (FROM PROBLEM 16) SO THAT IT FINDS THE SMALLEST OF 4 NUMBERS INSTEAD OF 3.

Correct solution:

```
    7.1 DEMAND A
    7.2 DEMAND B
    7.3 DEMAND C
    7.4 DEMAND D
    7.5 SET S = A
    7.6 SET S = B IF B < S
    7.7 SET S = C IF C < S
    7.8 SET S = D IF D < S
    7.9 TYPE "THE SMALLEST NUMBER IS"
    7.95 TYPE S
    DO PART 7
```

This problem requires modification of a similar program (used to find the smallest of three numbers) given as an example in the exercise that precedes this one in the curriculum.

L 15-18: WRITE A PROGRAM THAT TYPES THE LARGER OF 2 NUMBERS AND THEN THE SMALLER.

Hint:
REWRITE THE ABOVE PROGRAM SO THAT IT TYPES THE LARGER NUMBER FIRST, INSTEAD OF THE SMALLER.

Correct solution:

```
    1.1 DEMAND A
    1.2 DEMAND B
    1.3 TYPE A, B IF A > B
    1.4 TYPE B, A IF B > A
    1.5 TYPE A IF A = B
    DO PART 1
```

Again, only a slight modification of a sample program is required. The text for this problem includes an example of three steps similar to Steps 1.3, 1.4, and 1.5 above, with < in place of >. Solutions were considered correct even though they failed to provide for the case A = B.

L 15-21: WRITE A PROGRAM THAT WILL PRINT "SAME" IF ALL THREE NUMBERS X, Y, AND Z HAVE THE SAME SIGN. THE PROGRAM SHOULD PRINT "DIFFERENT" IF THE NUMBERS DO NOT ALL HAVE THE SAME SIGN.

Hint:
CAN YOU USE A COMMAND LIKE THIS?

```
     SET A = 1 IF X > 0 AND Y > 0 AND Z > 0
```

Correct solution:

```
     9.1 DEMAND X
     9.2 DEMAND Y
     9.3 DEMAND Z
     9.4 SET A = 0
     9.5 SET A = 1 IF X > 0 AND Y > 0 AND Z > 0
     9.6 SET A = 1 IF X < 0 AND Y < 0 AND Z < 0
     9.7 SET A = 1 IF X = 0 AND Y = 0 AND Z = 0
     9.8 TYPE "SAME" IF A = 1
     9.9 TYPE "DIFFERENT" IF A = 0
     DO PART 9
```

Preceding this problem is an example of a program that determines whether or not two numbers have the same sign. That problem is the first in the course in which conjunctions are used.

L 16-4: WRITE A PROGRAM THAT WILL DEMAND A RADIUS R AND THEN CALCULATE THE AREA OF A CIRCLE WITH THAT RADIUS. USE TWO PARTS, ONE FOR THE MAIN PROGRAM AND ONE FOR AN ERROR ROUTINE TO BE USED IF R IS NEGATIVE.

Hint #1:
THE AREA OF A CIRCLE IS 3.14159265 * R↑2.

Hint #2:
FIRST, GET THE RADIUS BY USING A DEMAND COMMAND.
SECOND, DECIDE WHETHER OR NOT TO GO TO THE ERROR ROUTINE.
THIRD, TYPE THE AREA.

Correct solution:

```
     3.1 DEMAND R
     3.2 TO PART 4 IF R < 0
     3.3 TYPE 3.14159265 * R↑2
     4.1 TYPE "A RADIUS CANNOT BE NEGATIVE"
     DO PART 3
```

This problem uses the branching command TO, which has just been introduced. An almost identical error routine is shown in a preceding problem.

L 16-6: WRITE A PROGRAM THAT WILL TYPE 3 NUMBERS A, B, AND C IN NUMERIC ORDER. USE SEVERAL PARTS TO MAKE IT EASIER TO CHECK EACH PART.

Hint #1:
PART 1 SHOULD FIND THE SMALLEST OF A, B, AND C, AND THEN
...BRANCH TO PART 2 IF A IS SMALLEST.
...BRANCH TO PART 3 IF B IS SMALLEST.
...BRANCH TO PART 4 IF C IS SMALLEST.

Hint #2:
PART 2 SHOULD BE USED IF A IS THE SMALLEST OF A, B, AND C.
PART 2 SHOULD DECIDE WHICH OF B AND C IS THE SMALLEST, ETC.

Correct solution:
```
    1.1 DEMAND A
    1.2 DEMAND B
    1.3 DEMAND C
    1.4 TO PART 2 IF A <=  B AND A <= C
    1.5 TO PART 3 IF B <=  A AND B <= C
    1.6 TYPE C, A, B IF A <= B
    1.7 TYPE C, B, A IF B > A
    2.1 TYPE A, B, C IF B <= C
    2.2 TYPE A, C, B IF C > B
    3.1 TYPE B, A, C IF A <= C
    3.2 TYPE B, C, A IF C > A
    DO PART 1
```

This problem is one of the longest and perhaps the most difficult in the entire course. The student has used TO in only one other problem (L 16-4 above), and no similar program is shown in the lesson. In grading the solutions to this problem the programs were expected to function only for unequal values of A, B, and C.

L 23-7: WRITE A PROGRAM THAT WILL PRINT THE MULTIPLICATION TABLE UP TO 5 TIMES 5.

Hint:
THE PROGRAM SHOULD PRINT SOMETHING LIKE THIS:

```
 MULTIPLICATION TABLE
1    2    3    4    5
2    4    6    8    10
3    6    9    12   15
4    8    12   16   20
5    10   15   20   25
```

Correct solution:

```
    4.1 SET X = 1
    4.2 TYPE X, X*2, X*3, X*4, X*5 IN FORM 7
    4.3 SET X = X + 1
    4.4 TO STEP 4.2 IF X <= 5
    FORM 7:
    ←← ←← ←← ←←← ←←.
    DO PART 4
```

This is the first program to use a loop (introduced in Lesson 23), and it was anticipated as a difficult problem, because no similar program was shown previously. The student is allowed to print results in linear form rather than in the tabular form shown above.

L 24-11: WRITE A PROGRAM THAT WILL DEMAND A VALUE FOR N, AND WILL THEN START AT 1 AND COUNT UP TO N.

FOR EXAMPLE, IF YOU GAVE 7 AS THE VALUE FOR N, THE PROGRAM SHOULD TYPE

```
    1
    2
    3
    4
    5
    6
    7
```

Hint:
THIS PROGRAM IS THE SIMPLEST POSSIBLE KIND OF LOOPING PROGRAM.
IF YOU CANNOT FIGURE OUT HOW TO DO IT, YOU HAD BETTER GO BACK TO THE BEGINNING OF LESSON 23 AND READ THE EXAMPLES VERY CAREFULLY.

Correct solution:

```
    6.1 DEMAND N
    6.2 SET C = 1
    6.3 TYPE C
    6.4 SET C = C + 1
    6.5 TO STEP 6.3 IF C <= N
    DO PART 6
```

The use of loops iterated a variable number of times is discussed in Lesson 24, the second lesson on loops, and this simple problem (with the unhelpful hint) is the first programming problem in the lesson.

L 25-8: HERE IS A PROGRAM WITH A LOOP:

```
     3.1 SET C = 1
     3.2 TYPE 1/C
     3.3 SET C = C + 1
     3.4 TO STEP 3.2 IF C < 8
```

REWRITE THE PROGRAM SO THAT YOU CAN USE A "FOR" CLAUSE.

Correct solution:

```
     1.1 TYPE 1/C
     DO STEP 1.1 FOR C = 1(1)7
```

The equivalence of two methods of iterated execution is discussed in Lesson 25, and this is the first programming problem requiring such a transformation.

L 26-5: WRITE A PROGRAM TO CONVERT INCHES TO FEET AND INCHES. USE A "DEMAND" COMMAND IN A LOOP.

Correct solution:

```
     4.1 DEMAND I
     4.2 SET F = IP(I/12)
     4.3 SET I = I - 12*F
     4.4 TYPE F, I IN FORM 4
     4.5 TO STEP 4.1
     FORM 4:
     ←←← FEET, ←← INCHES
     DO PART 4
```

Since the execution of a program containing a DEMAND command can be halted by answering the DEMAND with a carriage return, seemingly endless loops that incorporate DEMAND's are acceptable in AID programming. They are explained in Lesson 26.

L 29-19: WRITE A PROGRAM TO FIND WHICH OF THREE NUMBERS A, B, AND C IS CLOSEST TO 13/17.

Hint:
USE THE ABSOLUTE VALUE TO FIND THE DISTANCE. FIRST FIND WHETHER A OR B IS CLOSER TO 13/17. THEN FIND IF THAT ONE OR C IS CLOSER.

Correct solution:

```
    1.1 DEMAND A
    1.2 DEMAND B
    1.3 DEMAND C
    1.4 TO PART 2 IF !B - 13/17! < = !A - 13/17!
    1.5 TO PART 3 IF !C - 13/17! < = !A - 13/17!
    1.6 TYPE "A IS CLOSEST TO 13/17"
    2.1 TO PART 3 IF !C - 13/17! < = !B - 13/17!
    2.2 TYPE "B IS CLOSEST TO 13/17"
    3.1 TYPE "C IS CLOSEST TO 13/17"
    DO PART 1
```

Lesson 29 introduces the AID notation for absolute value, !x!, and discusses the use of absolute value for finding distances between points on the number line. No program similar to the above is used in examples. This problem is probably one of the most difficult in the course, primarily because the obvious approach (using conjunctions) produces commands that are too long to be correct AID commands. Solutions need not provide for the possibility that two points are at the same distance from the fixed point (in fact, the 'correct' solution shown above does not do this).

L 32-5: SET L EQUAL TO THIS LIST OF NUMBERS:
    1, 7, 14, 2, 5, 21
WRITE A PROGRAM THAT WILL TYPE ALL OF THE NUMBERS AND GIVE THEIR SUM. THEN CHANGE THE LIST TO THE FOLLOWING AND RUN THE PROGRAM AGAIN.
    5, 50, 100, 0, 1, 2

Hint #1:
USE ONE PROGRAM WITH A DEMAND COMMAND TO SET L EQUAL TO THE LIST. USE ANOTHER PROGRAM TO TYPE THE NUMBERS AND GIVE THEIR SUM. GET ANOTHER HINT IF YOU NEED MORE HELP.

Hint #2:
AT THE END OF THE LAST PROBLEM THERE IS A PROGRAM TO SET THE VALUES OF A LIST. THERE ARE OTHER WAYS TO DO IT, AND YOU MAY TRY ANYTHING YOU LIKE. USE A MAIN PROGRAM AND A SUBROUTINE FOR THE REST OF THE PROBLEM. THERE IS ANOTHER HINT IF YOU NEED IT.

Hint #3:
IN THE MAIN PROGRAM -- THE COMMAND TO REPEAT THE SUBROUTINE FOR EACH NUMBER IN THE LIST; THE COMMAND TO PRINT THE SUM.
IN THE SUBROUTINE -- THE COMMAND TO PRINT A NUMBER IN THE LIST; THE COMMAND TO ADD THAT NUMBER TO THE SUM.

Correct solution:

```
    2.1 DEMAND X
    2.2 SET L(I) = X
    DO PART 2 FOR I = 1(1)5
    3.1 SET S = 0
    3.2 DO PART 4 FOR I = 1(1)6
    3.3 TYPE S
    4.1 TYPE L(I)
    4.2 SET S = S + L(I)
    DO PART 3
```

This is the first programming problem in Lesson 32, which introduces lists and indexed variables. This problem is quite difficult, since the applicable strategies other than for input are not discussed before this problem is given.

L 32-8: WRITE A PROGRAM TO FIND THE AVERAGE OF THE NUMBERS IN A LIST OF TEN NUMBERS. TEST YOUR PROGRAM ON THESE TWO LISTS:

A. -10, 0, 1, 5, -3, 28, 17, 6, 11, -7
B. -.4, 2.5, 3.1, -5.8, 0, 7.1, 4, 8.9, .2, 3.1

Hint:
AVERAGE = SUM OF VALUES IN LIST/ NUMBER OF VALUES IN LIST.

Correct solution:

```
    3.1 DEMAND X
    3.2 SET L(I) = X
    DO PART 3 FOR I = 1(1)10
    4.1 SET S = 0
    4.2 DO STEP 5.1 FOR I = 1(1)10
    4.3 TYPE S/10
    5.1 SET S = S + L(I)
    DO PART 4
```

This problem should not be difficult for most students, since the preceding exercise shows an example of a program that computes the average of the numbers in a list of five numbers.

L 32-19: WRITE A PROGRAM TO FIND AND PRINT ALL THE NUMBERS LESS THAN 30 IN A LIST OF 10 NUMBERS. TEST YOUR PROGRAM ON THIS LIST:

10, 40, 39, 19, 28, 31, 30, 29.999, 16, 37

Correct solution:

```
    5.1 DEMAND X
    5.2 SET L(I) = X
    DO PART 5 FOR I = 1(1)10
    1.1 DO PART 2 FOR I = 1(-1)10
    2.1 TYPE L(I) IF L(I) < 30
    DO PART 1
```

This program is simpler than the preceding one (L 32-8), but no model is given in the lesson. Solutions that use $\leq$ in place of $<$ were considered incorrect.

This concludes the list of programming problems used in this study. None of the anticipated solutions are lengthy programs; the longest contains 12 commands and most require two to five commands. Although many of these problems appear simple, the students did not find them so. For this reason, solutions were not graded strictly; in some cases programs were considered correct even though they contained discontinuities not present in the expected solutions (for example, failure to account for the case X=Y in Problem L 15-15).

I turn now to a discussion of the construction of expected correct solutions. The criteria used for this task were not completely objective. The rules followed are listed in the order in which they were applied.

1. Only lexical elements and grammatical constructions that had been previously taught were allowed.

2. If a correct program or part of a program was shown in the problem statement, or in one of the hints, it was used.

3. If an applicable strategy was mentioned in the problem statement, or in one of the hints, that strategy was used.

4. If a similar program or part of a similar program was shown in the problem statement, or in one of the preceding four exercises, it was adopted, provided rules 2 and 3 were not violated.

5. If an applicable new AID construction was introduced in the current lesson, it was used provided none of the above rules were violated.

6. Of the correct solutions that satisfied the above requirements, the shortest was chosen. The shortest solution is defined as that which requires the fewest characters for the program and the commands necessary to execute it four times.

The expected correct solutions obtained by these rules are neither the shortest nor the most efficient solutions, and in a number of cases students produced more elegant programs. How well students' solutions were predicted is shown in Table 1. For 15 problems the expected solution was the most common correct solution; however, this was not always the case, and for five problems, no student gave the anticipated solution. A student's solution was considered equivalent to the expected solution if it differed by no more than optional spaces, names for variables, and step numbers. (This conception of program equivalence is the first one discussed in Chapter VII and a more precise definition is given there.).

Table 1

Comparison of Expected with Student-written Correct Solutions

Problems for which the expected solution		
was most frequent	occurred but was not most frequent	did not occur
L 5-30	L 8-9	L 11-11
L 8-27	L 8-28	L 15-21
L 9-3	L 9-8	L 29-19
L 10-12	L 15-18	L 32-8
L 10-19	L 26-5	L 32-19
L 12-4		
L 13-29		
L 15-15		
L 15-17		
L 16-4		
L 16-6 *		
L 23-7 **		
L 24-11		
L 25-8		
L 32-5		

* The expected solution was given by only one student but no other correct solution occurred more frequently.

** The expected solution was given by two students.

CHAPTER III

Description of the Data

The data consists of the (recorded) work performed by 40 students on 25 programming problems. Students worked a total of 747 problems. The distribution of number of problems attempted is shown in Figure 1. The number ranges from five to 25, with a mean of 19. Because the 1971-72 AID course was student-controlled, students were permitted to work problems in any order or to skip problems; thus, some students who completed all 32 lessons did not attempt every problem. In addition, several students did not complete 32 lessons, resulting in a steady decline from the first problem to the last in the number of students attempting a problem. This distribution--the number of students attempting each problem--is shown in Figure 2. The number of attempted solutions for a problem ranged from 14 to 38, with a mean of 30. Because of the high variance in the number of problems attempted by each student and in the number of students who attempted each problem, most of the statistics cited hereafter are given as proportions.

The fact that a student attempted a problem tells us little about how much work he did or how close he came to solving the problem. The correctness of students' solutions is discussed in the next chapter. A good indicator of the amount of effort expended is the number of AID commands typed while attempting to solve a problem. A total of 7063 commands were typed by all students. The number of commands typed for a problem ranged from 1 to 72, with an average of 7.1. The average number of commands typed for each problem is shown in Table 2.

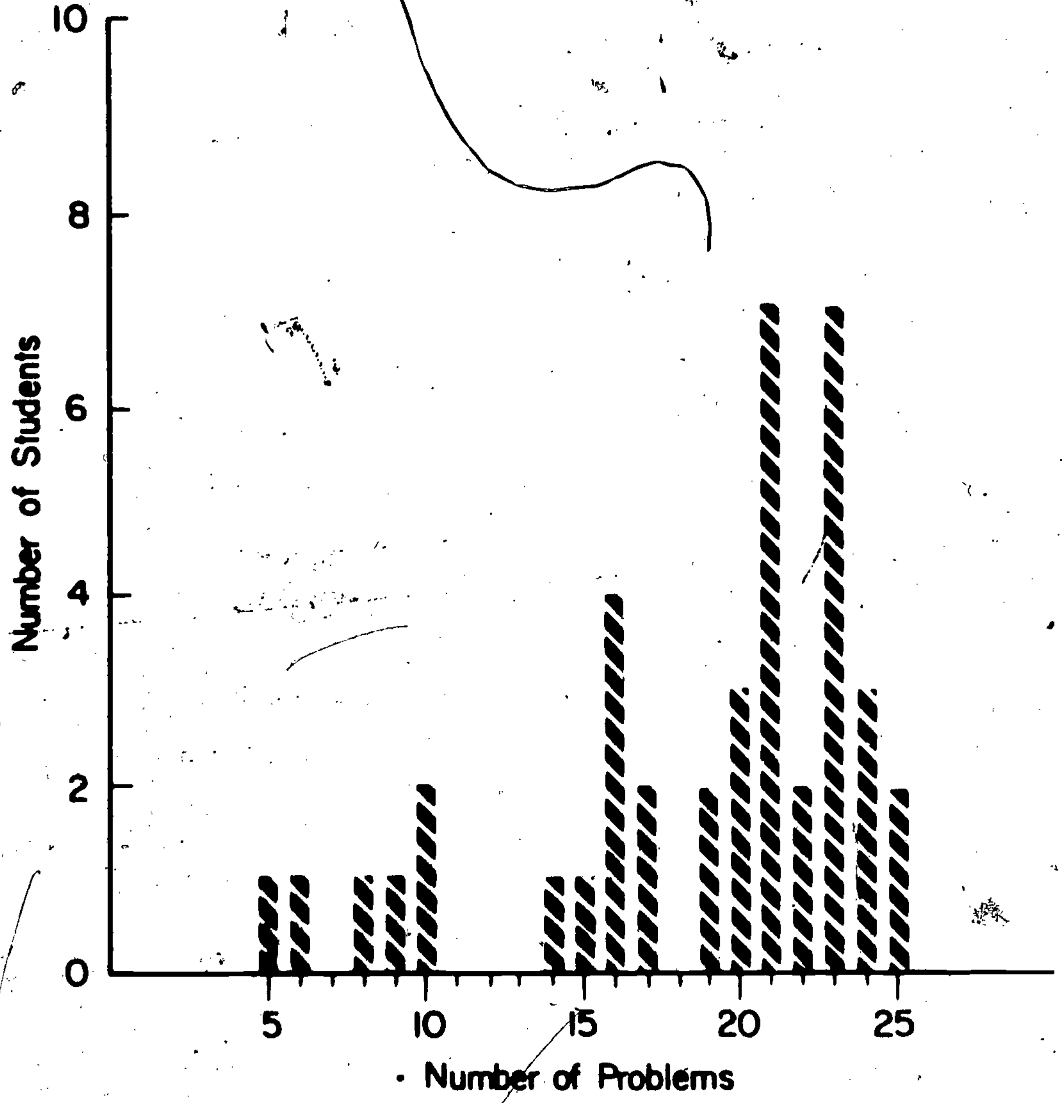

Figure 1. Distribution of number of problems attempted.

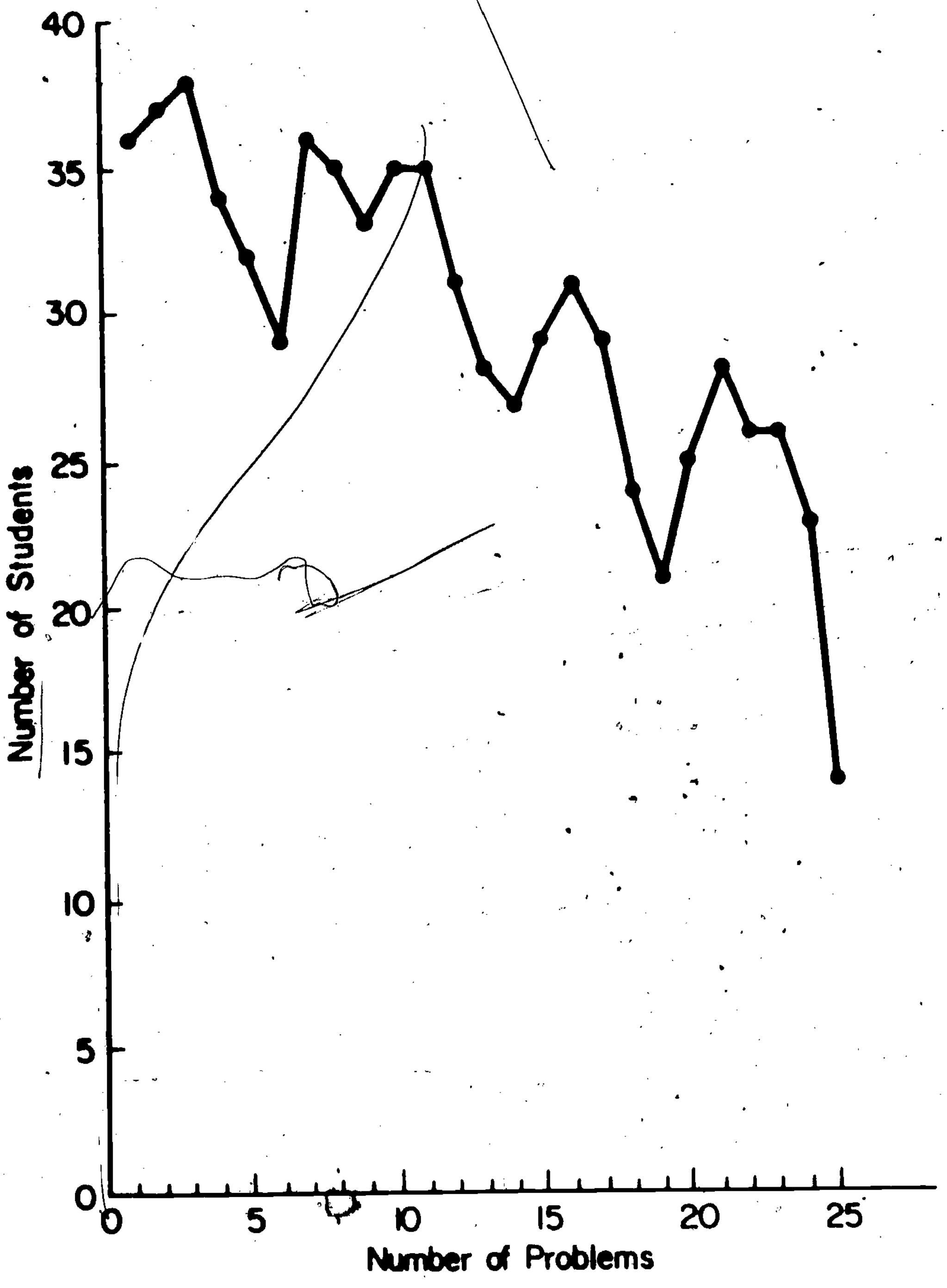

Figure 2. Number of students attempting each problem.

Table 2

Comparisons of Number of Commands Typed and Executed for Observed and Anticipated Solutions

Problem number	Number of Commands: Typed in student solutions (average)	In anticipated correct solutions	Executed in student solutions (average)
L5-30	3.7	2	3.5
L8-9	3.2	4	2.9
L8-27	4.5	2	3.7
L8-28	5.4	2	4.6
L9-3	7.1	2	5.3
L9-8	5.4	2	4.4
L10-12	2.9	2	2.3
L10-19	3.3	2	2.8
L11-11	7.7	5	5.9
L12-4	7.8	5	6.8
L13-29	6.6	3	4.4
L15-15	11.2	5	7.1
L15-17	13.3	11	10.3
L15-18	9.1	6	7.1
L15-21	15.0	10	11.8
L16-4	9.5	5	7.4
L16-6	22.6	12	14.3
L23-7	14.2	6	11.7
L24-11	10.6	6	8.2
L25-8	5.0	2	3.6
L26-5	10.1	7	8.0
L29-19	14.2	10	7.4
L32-5	23.3	9	14.7
L32-8	23.5	8	15.7
L32-19	16.4	6	12.0
	Mean: 10.22 S.D.: 6.26	Mean: 5.36 S.D.: 3.17	Mean: 7.44 S.D.: 3.99

r = .834 r = .821

As one would expect, the average number of commands typed per problem varies considerably from one problem to the next. As shown in Table 2, the average for Problem L8-9 is 3.2 lines, while the average for L32-8 is 23.5 lines. A comparison between the average number of commands typed and the number of commands used in the anticipated correct solutions is shown in Table 2. The number of commands typed is consistently greater than, in fact almost double, the number of commands in the anticipated solutions. One must conclude that the students' attempts to solve these problems were not cursory efforts. The correlation between expected and observed values is quite good; r = .834.

Perhaps a more useful measure of the amount of effort expended than the number of commands typed is the number of commands executed. Of the 7063 typed commands, 5177 were executed. Thus, 1886 of the typed commands--26 percent of the total--were unused, either because no attempt was made to execute them or because they contained errors that prevented their execution. We see in Table 2 that the number of commands in the anticipated correct solutions is less than the number executed, although the difference is not as great as that for typed commands. There are, however, three problems for which the average number of executed commands is less than the number used in the anticipated solution. One would expect a higher correlation for executed commands than for typed commands, but it is slightly lower, .821 compared with .834.

We can characterize the commands that constitute the data by looking at their function as programming commands, which may be done in two ways. First, we can classify commands according to whether they are direct (immediately executed) or indirect (stored) commands. Second, we can classify commands according to the AID verb used. The number of occurrences of the different kinds of commands, using both methods of classification, is shown in Table 3. A command does not appear in the data before it is introduced in the curriculum; that is, no indirect commands are used before Lesson 10, no DEMAND commands are used before Lesson 12, etc. With the exception of the LET command, which is used heavily in earlier problems and less in later problems, the number of occurrences of a given kind of command remains fairly constant after the command's introduction (although we see some large fluctuations from one problem to the next). Thus, though the total number of commands tends to increase with lesson number, the increase is due to a greater number of different commands used, rather than increased frequency of use. This is partly due to the nature of the curriculum, in which an effort was made to arrange problems and lessons so that a command once introduced was used frequently thereafter. That this did not occur with LET indicated a weakness in the curriculum, which was subsequently corrected.

We turn now to a comparison of the proportions of types of commands observed in the data and the proportions used in the anticipated correct solutions as shown in Table 4. Looking first at the classification by verb (second part of Table 4) we see that the anticipated solutions do not contain any occurrences of DELETE or of the file commands, and therefore they cannot serve as predictors of the number of occurrences of these kinds of commands. Even so, the match between predicted and observed is extremely good. The correlation coefficient is .958. The only two marked discrepancies are for SET and DEMAND; there is a higher proportion of SET commands in the data than in the expected solutions

Table 3

Number of Occurrences of Different Kinds of Commands by Problem

Problem number	Number of direct commands	Number of indirect commands	Classification by Verb: TYPE	SET	DEMAND	TO	DO	LET	DELETE	FORM	File commands*	Unidentifiable	Total
L5-30	134		60	71					2			1	134
L8-9	119		78					40				1	119
L8-27	172		72	48				48	2			2	172
L8-28	184		105	29				40	2			8	184
L9-3	228		125	38				55	3			7	228
L9-8	183		108	31				40	1			3	183
L10-12	60	46	51	4			42	5				4	106
L10-19	73	42	54	1			49	2	5			4	115
L11-11	80	173	100	69			57	7	11			9	253
L12-4	99	173	55	6	128		72	2	5			4	272
L13-29	107	124	62	22	37	1	70	24	8			7	231
L15-15	173	174	119	57	34	1	109	3	10			14	347
L15-17	93	280	62	127	117	1	57		2			7	373
L15-18	72	173	103	27	46	1	57		6			5	245
L15-21	108	327	86	152	99		82	3	4			9	435
L16-4	95	199	95	13	46	45	74	1	7	1		12	294
L16-6	147	509	187	95	109	135	85		28	3	4	10	656
L23-7	131	210	93	88	3	39	61	6	5	32	8	6	341

Table 3 (cont'd)

Problem number	Number of direct commands	Number of indirect commands	TYPE	SET	DEMAND	TO	DO	LET	DELETE	FORM	File commands*	Unidentifiable	Total
L24-11	71	151	40	55	26	34	47	3	4	5	1	7	222
L25-8	56	69	41	17	1	6	51		2	1	3	3	125
L26-5	104	178	70	63	33	21	51	1	3	30		10	282
L29-19	139	230	132	73	57	18	38		5	22	6	18	369
L32-5	271	336	124	142	44	33	151	15	44	4	20	30	607
L32-8	217	324	74	204	57	7	140	2	13		13	31	541
L32-19	88	141	49	66	17	9	57	3	13	3	4	8	229
Totals	3204	3859	2145	1498	854	351	1350	300	185	101	59	220	7063
	45.4%	54.6%	30.4%	21.2%	12.1%	5.0%	19.1%	4.2%	2.6%	1.4%	0.8%	3.1%	

*The file commands are the verbs USE, FILE, RECALL, and DISCARD.

Table 4

Comparison of Observed and Expected Proportions

of Different Kinds of Commands

Kind of Command	Number of Occurrences		Proportion	
	Observed*	Expected**	Observed	Expected
Direct	80.1	38	45.4%	28.4%
Indirect	96.5	96	54.6%	71.6%
Total	176.6	134		
TYPE	53.6	41	30.4%	30.6%
SET	37.4	25	21.2%	18.7%
DEMAND	21.4	27	12.1%	20.1%
TO	8.8	9	5.0%	6.7%
DO	33.8	25	19.1%	18.7%
LET	7.5	5	4.2%	3.7%
DELETE	4.6		2.6%	
FORM	2.5	2	1.4%	1.5%
File commands***	1.5		0.8%	
Unidentifiable	5.5		3.1%	
Total	176.6	134		
Mean	17.7	13.4		
S.D.	18.1	14.8		
	r = .958			

*The observed number of occurrences of the different kinds of commands is the total number of such occurrences divided by 40 (the number of students who contributed to the data).

**The expected number of occurrences of the different kinds of commands are taken from the expected correct solutions listed in Chapter II.

***USE, FILE, RECALL, and DISCARD

and a lower proportion of DEMAND commands. One reason for this is that students sometimes used direct SET commands as a means of input whereas the anticipated solutions used DEMAND. Another reason is that students undoubtedly made fewer errors in DEMAND commands, the simplest of the AID commands both in syntax and in semantics, and therefore retyped them less frequently. A study either of the proportions of different kinds of commands used in correct solutions or of the distribution of errors over command types would examine this hypothesis. Neither of these studies has yet been undertaken.

Looking at the comparison of the proportions of direct and indirect commands (also shown in Table 4), we see that the anticipated correct solutions do not function well as predictors of the observed proportions; students used a much higher proportion of direct commands than did the anticipated solutions. This can be explained partly by the fact that direct commands play a larger role in debugging than in writing programs, and the anticipated solutions cannot be expected to serve as predictors of commands used for debugging purposes. Another reason for the high proportion of direct commands in students' work is that students frequently omit the step number in what was intended as an indirect command. Since the criterion for classifying a command as either direct or indirect is the absence or presence of a step number, these erroneous commands were incorrectly classified.

CHAPTER IV

Distribution of Correct Solutions

Perhaps the single most important question to be answered by this study is: How many of the students solved the problems? It is to this question that this chapter is addressed.

In order to answer the question, one must first establish criteria for correctness, not a trivial task for programming problems. We could beg the question by referring the reader to Appendix A (which contains a list of all the correct solutions found in the data under consideration). Any solution in that list is correct. Thus, membership in the list is a sufficient condition for correctness, but not a necessary one. Rather than give a complete and exhaustive list of the criteria used in grading students' work, we will give an informal description of the attributes we looked for.

First, each correct program must perform a 'minimal' function. For most problems this function is defined by the correct solution listed in Chapter II, but for a few problems, the minimal function is a subset of that anticipated function. For example, for Problem L15-15, which asked for a program that would find the smaller of two numbers, the function defined by the anticipated correct solution is

$$f_A(x,y) = \begin{cases} x & \text{if } x < y \\ y & \text{if } x > y \end{cases} .$$

In grading students' work for this problem, the minimal function used was

$$f_M(x,y) = \begin{cases} x & \text{if } x < y \\ x & \text{if } x > y \end{cases}.$$

The domain of f_M excludes pairs (x, x), and hence, f_M is a proper subset of f_A. The comments given on grading in Chapter II, along with the correct solutions listed there, are sufficient to imply the minimal function that was used for each problem, so a complete list of minimal functions is not given here.

In one case (Problem L5-30) students were asked merely to compute three numerical results, and any method (other than computation by hand) that produced the three correct values was considered correct, so that the minimal function for this problem contains only these pairs. For all other problems a solution was not considered correct unless it was a general solution; that is, the domain of the minimal function is quite large. In a few instances, students' programs defined functions that included the anticipated function as well as the minimal function. In other words, the students' solutions were better than the minimal one; for an example of this, see Solution I_9 to Problem 23-7 (Appendix A).

The computational algorithm used by the student could be defined either as a stored program or as a user-defined function, and in general, the student used the same device as in the anticipated solutions. In either case, the students' solutions were required to print values as well as to compute them.

We have been discussing functions defined by programs as if they were real-valued functions. In fact, these functions ordinarily have as values text strings in which numeric values may or may not be imbedded.

Consider, for example, Problem L15-21, which asks for a program that will print either 'SAME' or 'DIFFERENT' depending upon a comparison of the signs of three numbers. Or, as another example, Problem L16-4 requires a subroutine to print an error message if a negative value is given as the radius of the circle. Problem L29-19 also requires text output. For these three problems a student's program was considered correct if it typed text with the appropriate content. Thus, for Problem L16-4, these error messages would all be considered equivalent and correct:

```
A RADIUS CANNOT BE BE NEGATIVE.
DON'T USE NEGATIVE NUMBERS.
YOU'RE NUTS!
```

Such decisions about equivalence of text are, of course, easy in hand grading but present great difficulties to an automated procedure for grading programs. Other than the three problems just mentioned, the minimal function used in grading did not include text. However, students' programs frequently provided for more than the minimal output. Often this was done by printing input values as well as output values. For example, one program to convert inches to feet and inches printed the result in the form

```
27 INCHES EQUALS 2 FEET AND 3 INCHES
```

rather than the simpler

```
2 FEET AND 3 INCHES
```

used by most programs. In grading, the context of the output values was ignored if it was not required by the minimal function.

In addition to computing and printing correct values, students' solutions were also required to handle input reasonably. Input in AID programming can be managed in two ways. Using DEMAND commands, a program can request the input data it needs, at execution time. A second method is to store data in the program before execution, by means of direct SET commands, a FOR modifier, or an auxiliary program that uses SET or DEMAND commands. Using either method, a student's solution was not judged to be correct unless it was executed correctly; for those problems that specified input values, the student's solution was considered correct only if he executed his program successfully for each specified value.

For certain problems, additional criteria of correctness were imposed. A problem statement might contain explicit instructions to use a specified kind of command or programming structure; for example, Problem L8-9 requires a LET command as does Problem L8-27; Problem L26-5 requires a loop incorporating a DEMAND command. Requirements implied but not explicitly stated in a problem statement are not taken as absolute, however, Thus, Problem L23-7, which asks for a program to print part of the multiplication table, did not require the output in tabular form, though the use of the word 'table' implied that it should.

For a few cases the standards described above proved inadequate in some way, primarily for the last three problems, which required the use of indexed variables. Solutions to such problems must be studied in much more depth and with more data before strategies for an automated program check can be devised.

In checking for correct solutions, all trials for the first encounter with a problem were inspected until a correct solution was found. Table 5 summarizes the performance on each problem, showing the number of correct solutions for the first trial, and the number correct

Table 5

Comparison of Performance on First Trial and All Trials

Problem number	Number of Correct Solutions		Proportion Correct*
	First trial	All trials	First trial
L5-30	17	35	.47
L8-9	31	36	.84
L8-27	25	29	.66
L8-28	21	28	.62
L9-3	19	29	.59
L9-8	12	19	.35
L10-12	27	33	.75
L10-19	30	34	.86
L11-11	22	26	.67
L12-4	28	34	.80
L13-29	21	21	.60
L15-15	15	19	.48
L15-17	18	19	.64
L15-18	11	11	.41
L15-21	20	22	.69
L16-4	24	25	.77
L16-6	7	9	.24
L23-7	12	16	.50
L24-11	8	8	.38
L25-8	18	18	.72
L26-5	15	17	.54
L29-19	3	3	.12
L32-5	7	7	.27
L32-8	10	10	.43
L32-19	6	7	.43
	427	515	

*Uses number of students attempting problem as denominator.

over all trials. Of the 747 attempted solutions, 427 (57%) were correct on the first trial and 515 (69%) were correct on some trial. The differences between performance on first and subsequent trials is not great, except for three problems (L5-30, L9-3, and L9-8) in which the second figure is 50% higher than the first. The score for each student was computed for first trials; the mean of these scores is 56%, with a range from 10% to 92%. The distribution of students' scores is shown in Figure 3. Two conclusions can be drawn--one, insofar as variance is an indicator, this set of programming problems was well chosen as a test, and two, the students did not find these problems easy. Although the performance of these same students on other exercises in the AID course has not been analyzed in detail, the average scores on all exercises in the course is over 75%, considerably higher than the 57% for the set of programming problems considered here.

The proportion correct, shown in the third column of Table 5, is used in Chapter VI as the primary measure of problem difficulty. Comparing the proportions correct for different problems, we note a range of .12 to .86. The three most difficult problems by this criterion are L16-6, L29-19, and L32-5. Both L16-6 and L29-19 are logically complex problems requiring the use of several conditional branches. L32-5 is the first problem using indexed variables, and the fact that it is quite difficult probably indicates the inadequacy of the curriculum rather than the inherent difficulty of the problem. A more detailed study of problem difficulty is pursued in Chapter VI.

Rather than judging solutions by a simple correct-incorrect scheme, we found that some system of assigning partial credit was also desirable.

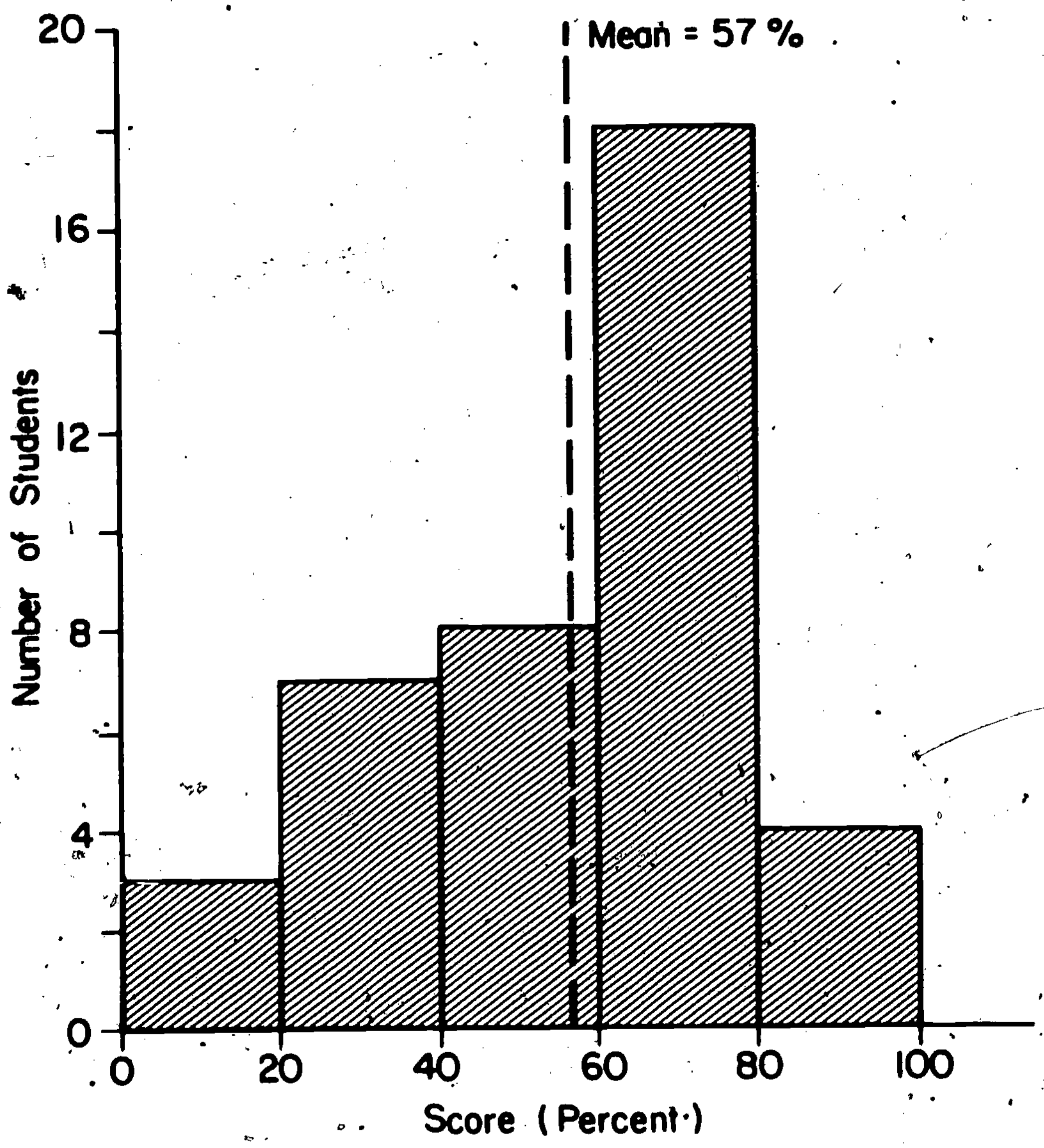

Figure 3. Distribution of students' scores.

The one used here is a simple count of the number of commands used in a correct or partially correct solution. This is not a completely satisfactory system, but it does have the virtue of providing a fairly objective measure. Using this measure of correctness, we were able to determine what proportion of the effort expended by students was useful effort. Table 6 shows the average number of commands used in correct and partially correct solutions for each problem. The criterion used in tallying the commands for Table 6 was not only that the command contribute to a correct solution, but also that the command be executed. Originally correct commands that were replaced by the student before execution did not contribute to these statistics, nor did commands that were stored but not executed.

In comparing the statistics from Table 6 with those in Table 2 (number of commands typed), note that less than half (3404/7063) the commands typed were used in correct or partially correct solutions. Students typed an average of 9.5 commands for the problems attempted, but only 4.6 commands contributed toward a correct solution. Looking at the totals for different problems, we see that for three problems (L29-19, L32-5, and L32-19) fewer than one-third of the typed commands contributed to correct solutions. Two of these problems are from Lesson 32, which introduces indexed variables, and again we attribute this to a weakness in the curriculum rather than to characteristics of the problems themselves. This supposition might be confirmed by comparisons with data from similar problems in the 1972-73 AID course in which the lessons on indexed variables were substantially revised.

Table 6

Number of Commands Used in Anticipated and Correct and Partially Correct Solutions

	Number of Commands	Average Number of Commands*
Problem number	Anticipated correct solution	Correct and partially correct solutions
L5-30	2	2.2
L8-9	4	2.6
L8-27	2	2.7
L8-28	2	3.1
L9-3	2	4.1
L9-8	2	4.0
L10-12	2	2.0
L10-19	2	2.1
L11-11	5	4.0
L12-4	5	4.7
L13-29	3	2.7
L15-15	5	3.8
L15-17	11	8.5
L15-18	6	4.8
L15-21	10	9.9
L16-4	5	5.8
L16-6	12	8.0
L23-7	6	6.2
L24-11	6	5.9
L25-8	2	1.7
L26-5	7	6.5
L29-19	10	3.7
L32-5	9	5.1
L32-8	8	10.5
L32-19	6	5.0

*Used 3404 typed commands.

A similar comparison can be made between the number of commands used in correct or partially correct solutions and the number of executed commands, which are shown in Table 2. This comparison is more meaningful than the preceding one because the commands tallied in Table 6 had to be executed, and thus we are comparing the number of executed commands that contributed toward a correct solution with the total number of executed commands. Of the commands that were executed, two-thirds (3404/5177) contributed toward a correct solution. For a detailed comparison, look again at the averages for the individual problems. We find that fewer than half of the executed commands contributed toward correct solutions for three problems: L25-8, L32-5, and L32-19. The last two of these are the same two (quite difficult) problems from Lesson 32 for which the discrepancy with commands typed was so marked. Interestingly, the other problem, L25-8, is one of the easiest problems in the set with a probability correct of .72. Looking at discrepancies at the other end of the scale, we find that 90% of the executed commands contributed to correct solutions for Problems L8-9 and L9-8. For L8-9, 84% of the students produced correct solutions on their first trial, but for L9-8 that figure is only 35%. Obviously no simple relation exists between these different measures and a more detailed analysis of the relationships is undertaken in Chapter VI.

In the correct-incorrect grading we allowed only completely correct solutions. However, relaxing these standards somewhat, we can define another variable, allowing as correct those programs that are correct up to algebraic expressions. By disregarding algebraic errors, we can obtain additional, possibly better, evidence of the programming difficulty represented by the different problems. The pertinent summary statistics for this variable are shown in Table 7. The number of solutions that were correct, except for algebraic errors, is shown separately from the total in order to emphasize the variance. The correlation between these two measures of proportion correct is quite high (r = .904). Notice that 15 students solved Problem L5-30 correctly, except for algebraic errors; this nearly equals the number who solved the problem completely correctly (17), and changes the proportion correct for that problem from .47 to .89. Other changes are less impressive, but several others are also substantial: the proportion correct for L9-8 changes from .35 to .56, for L10-12 from .78 to .92, and for L24-11 from .38 to .52. For nine of the problems, no change in proportion correct is achieved by modifying the definition of correctness.

In some cases the definition of the minimal function, discussed earlier, markedly affected the measures derived for proportion correct. The problems that would be most noticeably affected if the criteria were more stringent are L15-15, L15-18, L16-6, and L29-19, in all of which the students were allowed to ignore the possibility that different input variables might have equal values. For all four of these problems, fewer than half of the attempted solutions were graded correct, and this proportion would decrease substantially with any increase in the stringency of the criteria.

Table 7

Number of Solutions That Were Correct Except for Algebraic Errors (first trials)

Problem number	Number of solutions that were correct except for algebra	Number of solutions* that were correct or correct except for algebra	Proportion correct**
L5-30	15	32	.889
L8-9	2	33	.892
L8-27	2	27	.711
L8-28	4	25	.735
L9-3	1	20	.625
L9-8	7	19	.559
L10-12	5	32	.917
L10-19	2	32	.914
L11-11	0	22	.667
L12-4	5	33	.943
L13-29	0	21	.600
L15-15	1	16	.516
L15-17	0	18	.643
L15-18	0	11	.407
L15-21	0	20	.690
L16-4	2	26	.839
L16-6	1	8	.310
L23-7	0	12	.500
L24-11	3	11	.524
L25-8	3	21	.840
L26-5	1	16	.571
L29-19	1	4	.154
L32-5	0	7	.269

(continued)

Table 7 (cont'd)

Problem number	Number of solutions that were correct except for algebra	Number of solutions* that were correct or correct except for algebra	Proportion correct**
L32-8	0	10	.435
L32-19	0	6	.429

*Sum of the number of solutions that were correct except for algebraic errors (taken from the preceding column) and the number of solutions that were completely correct.

**The proportion correct for each problem is calculated by the formula: total number of solutions that were correct or correct except for algebra ÷ number of students who attempted the problems.

CHAPTER V

Errors

The primary reason for undertaking an analysis of errors was to provide variety in the means of measuring problem difficulty. In addition, a study of errors provides insights into student performance not obtained from examinations of proportion correct and distribution of correct solutions.

We considered mainly overt errors; errors of omission are not examined, except for failure to provide for program output. A student may fail to solve a problem correctly, but at the same time make no overt errors. If his work is correct but not complete, the student is not credited with either a correct solution or any overt errors. Even more dramatically, a student may produce a large number of commands that are correct in the sense that they contain no errors and yet may not be credited with any commands that contribute to a correct solution because his work has no identifiable relation to the problem he is supposedly solving. There were a number of such instances in the data. In some cases, the student was clearly working on a completely unrelated program, e.g., on a previous problem, perhaps, or even one of his own choosing (one student spent considerable time writing a game-playing program that had no relation to any of the problems in the course).

In this chapter we use two methods to derive statistics. In the first method, all errors regardless of their source are classified by type, and in the second, we show for each problem the number of students who made errors, though not the number of errors made by each student.

There were 1090 errors in the 7063 commands typed by students, with some commands containing more than one error. Of the 1090 errors, 740 (68%) were syntax errors and 350 (32%) were semantic errors. Note that because we are concerned only with overt errors the proportion of syntactic errors is probably overestimated. This method is more likely to fail to count semantic than syntactic errors.

Syntax errors were divided into seven major classes containing 22 sub-classes, and the distribution of errors into these classes is shown in Table 8. Format errors, which accounted for 12% of the total, are of four types.

1. Line too long (3.0%). AID commands must be contained within lines of 72 characters or less. If a typed line exceeds 72 characters, an error message is given by the interpreter.

2. Omitted space (5.9%). One or more spaces are required as delimiters after step numbers, after verbs, on both sides of IF, AS, FOR, etc.

3. Inserted space (1.6%). Spaces are not allowed before the left parenthesis in expressions like F(X) and F(3), where F is either a user-defined function or a standard AID function; all of the observed errors were of this type. Nor are spaces allowed before the left parenthesis in expressions like X(2) and L(1,4) where X and L are indexed variables, but there were no occurrences of this error.

4. Visible delimiter errors (1.5%). Visible delimiters such as commas and semicolons are required in specific commands. Some of the errors in this subclass were errors of omission and others were errors of substitution.

Table 8

Classification of Syntax Errors

Classification	Number of errors	Percent of total errors
I. Errors in Format		
A. Line too long	22	3.0
B. Omitted space	44	5.9
C. Inserted space	12	1.6
D. Delimiter error	11	1.5
Total	89	12.0
II. Transient Errors		
A. Typographical error	80	10.8
B. Probable typographical error	36	4.8
C. Incomplete command	137	18.5
Total	253	34.1
III. Errors in Verbs		
A. Omitted verb		
1. SET	33	4.5
2. Other	5	0.7
Total	38	5.2
B. TO or DEMAND used directly	30	4.0
C. Incorrect verb	5	0.7
Total	73	9.9

(continued)

Table 8 (cont'd)

Classification	Number of errors	Percent of total errors
IV. Errors in Arguments of Verbs		
A. Equation used in TYPE command	11	1.5
B. Omitted "STEP" or "PART"	24	3.2
C. Errors in algebraic expressions		
1. Unmatched parentheses or absolute value signs	15	2.0
2. Other	20	2.7
Total	35	4.7
D. Omitted quotation marks	7	1.0
Total	77	10.4
V. Errors in Multiple Form of Argument		
A. Used X(1,2,3) for X(1),X(2),X(3)	8	1.1
B. Used DEMAND or SET with multiple argument	23	3.1
C. Omitted second occurrence of "PART" or "STEP"	7	1.0
Total	38	5.2
VI. Errors in Modifiers		
A. Misplaced IF clause	6	0.8
B. Error in logical expression	7	1.0
C. Modifier used with wrong verb		
1. FOR with TYPE	16	2.1
2. Other	7	1.0
Total	23	3.1

(continued)

Table 8 (cont'd)

Classification	Number of errors	Percent of total errors
D. Used FOR with more than one variable	34	4.6
Total	70	9.5
VII. Miscellaneous	140	18.9
TOTAL	740	100.0

The next three subclasses are contained in the class called 'transient errors'. These are typographical and related errors, and accounted for 34.1% of all syntax errors.

5. Typographical errors (10.8%). A strict criterion was used for this subclass. The errors include (a) doubling of letters (PARRT for PART), but not doubling of nonalphabetic characters, (b) omitting of letters (DELTE for DELETE), but only for words containing at least three other letters in correct sequence so that the word could be identified unambiguously, (c) substituting L for 1, and (d) substituting any character for another character with an adjacent keyboard position, provided that both characters were not digits and that no other similar substitution resulted in an identifiable expression with a different semantic value (for example, FO can be taken as a typographic substitution for DO or for TO since F is adjacent to both T and D on the keyboard; hence, this error was not classified as a typographical error).

6. Probable typographical errors (4.8%). This class includes the typographical errors that did not satisfy the above criterion. Categorizing these errors was guided in part by the student's subsequent action. For example, if a student typed a line like

3.15 FO PART 7

and immediately replaced it (before execution) by

3.15 TO PART 7

the error was included here.

7. Incomplete commands (18.5%). If a line is an initial segment of some correct AID command, it was counted as an error in this class. In most instances, it appeared that the student changed his mind in

midstream and, rather than type an erase command to erase the line, he typed the RETURN key and then retyped the command making the desired correction. If these errors are considered momentary aberrations in the same sense as typographical errors, rather than as semantic errors, they account for 34% of all syntax errors. Together with subclasses (1) and (2), which are also transient errors, the total reaches 43%, a substantial portion of the syntax errors.

Errors in verbs, which constitute 9.9% of the syntax errors, are divided into three subclasses:

8. Omitted verbs (5.2%). Because this occurred much more frequently for SET than for all other verbs, this subclass was subdivided to emphasize that difference. Of the 5.2% of the syntax errors that are due to omitted verbs, 4.5% are omissions of SET. The SET command, unlike any other AID command, can be given without the verb but only when used directly. To illustrate this distinction,

```
X = 7
```

may be used in place of

```
SET X = 7,
```

but

```
3.5 X = 7
```

cannot replace

```
3.5 SET X = 7.
```

We shall see other evidence of such logical 'overgeneralization' again in this discussion.

9. TO or DEMAND used directly (4.0%). Of the commands taught in the first 32 lessons of the AID course, TO and DEMAND are the only two

that cannot be used directly. Although the error message is specific and unambiguous (DON'T GIVE THIS COMMAND DIRECTLY), there were 30 such errors. A reasonable explanation for many of these errors is that the student forgot to type a step number; this explanation is supported by the fact that the errors occurred more frequently in the first step of a program (with DEMAND) than elsewhere, giving one the impression that the student's concentration on the semantic structure of the program was sufficiently intense to preclude minor syntactic considerations. This kind of error is related to errors in the first subclass of semantic errors and is mentioned again when those errors are discussed.

10. Incorrect verbs (0.7%). We expected that there would be more errors due to incorrect verbs than the five found in the data. The incorrect verbs found include DELETE for DISCARD, PRINT for TYPE, etc. There was no evidence of misspellings other than typographical errors, which are not in this class.

The fourth category of syntax errors includes those made in arguments of verbs. These accounted for 10.4% of the syntax errors, somewhat more than the 9.9% for errors in verbs, and less than either the 12.0% for format errors or the 34.1% for transient errors.

11. Equation used as arguments for TYPE (1.5%). Technically, a command like

TYPE Y = 2 * X

is not a syntax error since logical expressions can be used as arguments for TYPE (and will return either TRUE or FALSE). However, this form of the TYPE command was not taught in the first 32 lessons, and other evidence of the data indicates that students were incorrectly using this command as a combination of

```
SET Y = 2 * X
TYPE Y.
```

All such instances caused an error message that an undefined variable had been used, so that if one takes a more rigid view of the classification scheme, these errors should be grouped into the first subclass of semantic errors. We felt that to do so would be misleading even though correct.

12. Omitted STEP or PART (3.2%). Some examples of these errors are:

```
DO 3.1      for  DO STEP 3.1
TO 4        for  TO PART 4
DELETE 5    for  DELETE PART 5.
```

(Since a command like DO 3.1 cannot be interpreted as other than DO STEP 3.1, one might reasonably fault the interpreter rather than the student. The same complaint can be made about many other errors described here.)

13. Errors in algebraic expression (4.7%). Only syntactic errors are included here. Semantic errors in algebraic expressions are discussed later. Nearly half (15 out of 35) of these errors were in grouping, described in Table 8 as 'unmatched parentheses or absolute value signs'. In many cases these errors had more the appearance of typographical than of conceptual errors, as in

```
TYPE F(3.5)).
```

Among the other errors in algebraic expression, one that occurred several times was the omission of the multiplication symbol *.

14. Omitted quotation marks (1.0%). Because of numerous use-mention errors that occurred during pilot testing of the AID course, we expected a higher rate of these errors. In fact, students used text strings more often and made fewer errors than we expected.

The next three subclasses of syntactic errors also occurred in arguments of verbs but they were errors in the forms of multiple arguments rather than single arguments.

15. Used X(1,2,3) for X(1),X(2),X(3) (1.1%). Errors of this kind usually occurred in TYPE commands:

```
TYPE F(10,20,30)  for  TYPE F(10),F(20),F(30).
```

These errors could have been classed with delimiter errors, but were so different from other delimiter errors that they were put into a separate class.

16. Used DEMAND or SET with multiple arguments (3.1%). The only two AID verbs that allow multiple arguments are TYPE and DELETE, so commands like

```
SET X = 1,2,3
```

and

```
DEMAND X,Y,Z
```

are in error. Apparently, the students overgeneralized the rule that allows multiple arguments and produced these reasonable but erroneous commands.

17. Omitted second occurrence of PART or STEP (1.0%). Some examples of these errors are:

```
TYPE PART 2;3,4
DELETE STEP 2.1,2.15,2.2
```

Also grouped with these are similar commands in which the words PART or STEP were pluralized:

```
TYPE STEPS 1.2, 1.3, 1.4
DELETE PARTS 10,11,12.
```

The next four subclasses contain errors found in AID modifiers.

18. Misplaced IF clause (0.8%). This kind of error, which occurred rarely, was caused by a transposition of the main clause and the conditional clause:

```
3.7 IF X > Y TYPE X.
```

This order of clauses is used in many other programming languages.

19. Error in logical expression (1.0%). Several of these errors resulted from attempts to use commas to indicate conjunctions:

```
TYPE A IF A < B,C,D.
```

20. Modifier used with wrong verb (3.1%). The most common of these errors (16 out of 23) resulted from an attempt to use FOR as a modifier of TYPE:

```
TYPE 3*X↑(1/2) FOR X = 1,2,3,4.
```

Since FOR can be used only with DO, this resulted in an error message. Other instances of errors in this class are the use of TIMES as a modifier for DEMAND, the use of AS as a modifier TYPE, etc.:

```
DEMAND X, 3 TIMES
TYPE R as "RADIUS".
```

21. Used FOR with more than one variable (4.6%). Most occurrences of this error were for Problem L15-15, and reflect an omission in the curriculum. The problem asked for a program that would type the larger of two numbers, and the partial model shown in the problem did not

include any clues to how the input of two variables could be managed. The students had little previous experience with multiple input but had used FOR on many occasions to input several values for a single variable. Some examples of the commands produced by students are

```
DO PART 2 FOR X = 1, Y = 2
DO PART 2 FOR X = 1; Y = 2
DO PART 2 FOR (X,Y) = (1,2).
```

Students tended to persist in these errors, typing the same command again, or a similar one, even after receiving an error message. With a warning that FOR cannot be used with more than one variable, these errors could probably have been avoided entirely. (Such a change was made in a subsequent revision of the curriculum.)

22. Miscellaneous (18.9%). The errors classed as miscellaneous are too varied to be simply characterized. However, a large portion of these are probably typographical and are of a transient nature (i.e., many of the errors were corrected before execution or after an error message was given). A few of these errors were caused by attempts to use text strings as one of several arguments for a TYPE command:

```
TYPE "THE AREA IS", A
```

This error was caused by a bug in the interpreter and no warning about it was given in the lessons.

The second main group of errors, the semantic errors, are also divided into classes and subclasses--in this case, 9 classes containing 16 subclasses. The distribution of semantic errors is shown in Table 9. Of the 1080 errors analyzed, 350 (32%) were semantic. Each subclass is described individually. The first four subclasses are in the class

Table 9

Classification of Semantic Errors

Classification	Number of errors	Percent of total errors
I. Errors in Use of Variables		
A. Real	106	30.3
B. Functions	9	2.6
C. Step or Part	20	5.7
D. Other	9	2.6
Total	144	41.2
II. Algebraic Errors		
A. Omitted parentheses	6	1.7
B. Incorrect operator	25	7.1
C. Other	66	18.9
Total	97	27.7
III. Errors in Logic		
A. Logical expressions	14	4.0
B. Sequence of execution	16	4.6
C. Other	2	0.6
Total	32	9.2
IV. Errors in Use of Dummy Variables	13	3.7
V. Confusion between LET and SET	6	1.7
VI. Confusion between STEP and PART	8	2.3
VII. Numerical Error	11	3.1
VIII. No Provision for Output	6	1.7
IX. Miscellaneous	33	9.4
TOTAL	350	100.0

titled "Errors in Use of Variables." The errors in the first three subclasses are 'reference errors', that is, errors caused by attempts to use undefined variables, modified functions, etc. Errors in the use of dummy variables are not included here but in Subclass 11. Altogether, errors in the use of variables, excluding dummy variables, accounted for 41.2% of the semantic errors.

1. Real variables (30.3%). These errors resulted from attempts to execute a command containing undefined real variables, unindexed or indexed, in an algebraic expression. A large number of these errors were caused by the inadvertent omission of a step number, which caused the commands to be executed immediately, rather than stored and executed later as intended. Thus, some of the errors in this class are closely related to some of the errors in Syntax Subclass 9.

2. Functions (2.6%). These errors resulted from attempts to use undefined functions. In several cases students used one name for the function when defining it and inadvertently used another in a later function call. Some of these errors, however, indicate a deeper conceptual misunderstanding; errors in which the name of the dummy variable was used as the function name in the function call are of this type.

3. Step or part (5.7%). These errors occurred when students attempted to execute, list, or delete an undefined step or part.

4. Other errors in the use of variables (2.6%). Most of these occurred with indexed variables.

All of the errors listed above caused error messages, and are thus closely related to the syntax errors. Also, like most syntax errors, most of these errors were immediately corrected by the students and are not indicative of any serious misconceptions. The remaining semantic errors are, for the most part, evidence of more fundamental misunderstandings.

The next three subclasses contain algebraic errors, other than syntax errors.

5. Omitted parentheses (1.7%). Most of these errors occurred in Problem L12-4, where students used expressions like A + B + C/3 to find the average of three numbers.

6. Incorrect operator (7.1%). Most of these errors occurred in Problem L5-30, where students used expressions like 6.9*.3937 instead of the correct 6.9/.3937.

7. Other algebraic errors (18.9%). This is the second most numerous subclass of the semantic errors, and the errors were varied. Many were the result of incorrect translations of algebraic expressions into AID notation and many others were the result of incorrect expressions that were correctly translated. The two problems with the most errors in this subclass were the only two problems that required the use of the standard AID function IP (integer part).

The next three subclasses contain errors in logic, which accounted for 9.2% of the semantic errors.

8. Errors in logical expressions (4.0%). These errors in forming conditional clauses occurred most frequently for Problem L24-11, which asked for a program to count from 1 to N, and the most common error was the use of < for <=, which caused the program to count to N - 1 rather than N.

9. Errors in sequence of execution (4.6%). There were fewer errors in sequence of execution than we anticipated; however, errors caused by the complete omission of a branching command were not tabulated. One error that occurred several times was of this form:

```
1.1 DEMAND X
1.2 TYPE F(X)
1.3 DO PART 1
```

If TO had been used in place of DO, this program would have been considered correct. As it is, it functions correctly for a large number of iterations (sufficient for most purposes) and fails only when the capacity of the push-down stack is exceeded. Since students were taught nothing about this feature of the interpreter, and since this program functioned correctly from the student's point of view, it might have been better not to have considered the DO command in error.

10. Other errors in logic (0.6%).

11. Errors in use of dummy variables (3.7%). Several of these errors occurred when the student changed the name of the dummy variable in the middle of a LET command:

```
LET F(X) = 3.14*R↑2.
```

All of the errors with dummy variables indicated a serious conceptual difficulty, which the curriculum did little to dispel.

12. Confusion between LET and SET (1.7%). As a rule, LET and SET cannot be interchanged, and certainly not in the ways they were used in the lessons. There were, however, several instances where LET was used correctly, but a SET command would have been preferable. These did not

count as errors, although it is likely that these students were confused about the difference between LET and SET.

13. Confusion between STEP and PART (2.3%). An example of the errors in this subclass is

```
DO PART 3.25.
```

14. Numerical errors (3.1%). Several numerical errors were probably typographical in nature.

15. No provision for output (1.7%). Again, more errors were expected than occurred.

16. Miscellaneous (9.4%).

The error analysis was undertaken more in the spirit of demonstrating a method of error analysis for use with similar data than as a definitive study of the kinds of errors students make in learning to program. With only 25 problems and 40 students, the data are too sparse to warrant viewing the statistics as more than indications of tendencies.

Some tendencies are clearly indicated, however. Typographical errors accounted for the largest part of the syntax errors, and reference errors (Subclasses 1, 2, and 3 of the semantic errors) accounted for the largest portion of the semantic errors. A sizable number of syntax errors are 'reasonable' errors, that is, commands that could have been interpreted correctly had the interpreter been prepared for them; most of these resulted from misapplying--or overextending--some existing syntactical rule. In the semantic errors the second most numerous subclass was the class of algebraic errors, which partially confirms a previous suspicion that students were more deficient in algebra than the curriculum assumes, and that the curriculum does not devote enough time to teaching underlying algebraic concepts.

Several summary statistics are presented in Table 10 for individual problems. For each problem, the totals of both syntactic and semantic errors and the ratios of these to the number of students who attempted each problem are shown. Some of these derived statistics are used in the next chapter as measures of problem difficulty.

In comparing the error rates for different problems it is clear that Problems L32-5 and L32-8 are difficult both syntactically and semantically; these two problems also appeared among the most difficult by other measures of difficulty.

The correlation between syntactic and semantic errors is not high (-.34) and the most striking discrepancy is for Problem L29-19 for which there were 70 syntax errors and only 8 semantic errors. This problem is also one that was mentioned as extremely difficult by the criteria used in Chapter IV.

The error analysis described above yielded some interesting results and pointed the way for future detailed studies of similar data. However, the results may be misleading because in classifying and counting errors, we used the occurrences of errors rather than of the number of students who made errors. One would expect some correlation between the two categories but it would be far from perfect for there were a number of cases in which a sizable number of errors were made by only a relatively few students. This was particularly striking when a student persisted in repeating an error many times even after receiving intervening error messages.

Table 10

Classification of Errors by Problem

	Type of Error			
	Syntax		Semantic	
Problem number	Number of errors	Errors per student	Number of errors	Errors per student
L5-30	11	0.31	23	0.64
L8-9	10	0.27	9	0.24
L8-27	21	0.55	12	0.32
L8-28	23	0.68	15	0.44
L9-3	26	0.81	18	0.56
L9-8	18	0.53	20	0.59
L10-10	9	0.25	13	0.36
L10-19	12	0.34	7	0.20
L11-11	19	0.58	8	0.24
L12-4	18	0.51	21	0.60
L13-29	22	0.63	17	0.49
L15-15	61	1.97	11	0.35
L15-17	36	1.29	11	0.39
L15-18	25	0.93	11	0.41
L15-21	32	1.10	9	0.31
L16-4	36	1.16	10	0.32
L16-16	54	1.86	8	0.28
L23-7	23	0.96	7	0.29
L24-11	21	1.00	19	0.90
L25-8	11	0.44	12	0.48
L26-5	35	1.25	19	0.68
L29-19	70	2.69	8	0.31
L32-5	75	2.88	30	1.15

(continued)

Table 10 (cont'd)

	Type of Error			
	Syntax		Semantic	
Problem number	Number of errors	Errors per student	Number of errors	Errors per student
L32-8	51	2.22	25	1.09
L32-19	21	1.50	7	0.50
TOTAL	740	1.07 (average errors per student per problem)	350	0.49 (average errors per student per problem)

CHAPTER VI

Problem Difficulty

In the last three chapters several measures of problem difficulty were discussed: the number of typed commands that contribute toward a correct solution, the proportion of students who produced correct solutions, the number of syntax errors, etc. In this chapter these and other measures are compared, and an attempt is made to account for the variance in problem difficulty.

Nineteen measures are defined below. All are measures of qualities presumed to be related in some way to problem difficulty. Some, like the rate of syntax errors, vary directly with problem difficulty, whereas others, like the proportion of students who produced correct solutions, vary inversely. The first three variables are measures of proportion correct, and the statistics are derived from those discussed in Chapter IV (Distribution of Correct Solutions). The next ten measures are based on errors; the values of these variables are found from the statistics discussed in Chapter V (Errors). There are five measures of the effort expended, using statistics from Chapters III and IV. The final measure, the proportion of students who attempted the problem, is evaluated from statistics given in Chapter II. The 19 measures are described below and the values for each problem are given in Table 11.

Proportion Correct. All three measures of proportion correct are ratios of the number of students who gave correct solutions to the number of students who attempted the problem. This definition of proportion correct is somewhat different from the definitions used by others.

Table 11

Values for Measures Related to Problem Difficulty

Prob. No.	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
L5-30	.472	.889	.972	.306	.639	.944	.082	.172	.254	2.091
L8-9	.838	.892	.973	.270	.243	.514	.084	.076	.160	.900
L8-27	.658	.711	.763	.553	.316	.868	.122	.070	.192	.571
L8-28	.618	.735	.824	.676	.441	1.118	.125	.082	.207	.652
L9-3	.594	.625	.906	.813	.563	1.375	.114	.079	.193	.692
L9-8	.353	.559	.559	.529	.588	1.118	.098	.109	.208	1.111
L10-12	.778	.917	.944	.083	.361	.444	.028	.123	.151	4.333
L10-19	.857	.914	.971	.343	.200	.543	.104	.061	.165	.583
L11-11	.667	.667	.788	.576	.273	.848	.075	.036	.111	.474
L12-4	.800	.943	.971	.514	.600	1.114	.066	.077	.143	1.167
L13-29	.600	.600	.600	.629	.486	1.114	.095	.074	.169	.773
L15-15	.484	.516	.613	1.968	.355	2.323	.176	.032	.207	.180
L15-17	.643	.643	.679	1.286	.393	1.679	.097	.029	.126	.306
L15-18	.407	.407	.407	.926	.407	1.333	.102	.045	.147	.440
L15-21	.690	.690	.759	1.103	.310	1.414	.074	.021	.094	.281
L16-4	.774	.839	.806	1.161	.323	1.484	.122	.034	.156	.278
L16-6	.276	.310	.345	1.862	.276	2.138	.082	.012	.095	.148
L23-7	.500	.500	.667	.958	.292	1.250	.067	.021	.088	.304
L24-11	.381	.524	.381	1.000	.905	1.905	.095	.086	.180	.905
L25-8	.720	.840	.720	.440	.480	.920	.088	.096	.184	1.091
L26-5	.536	.571	.607	1.250	.679	1.929	.124	.067	.191	.543
L29-19	.115	.154	.115	2.692	.115	2.808	.190	.008	.198	.043
L32-5	.269	.269	.269	2.885	1.154	4.038	.124	.049	.173	.400
L32-8	.435	.435	.435	2.217	1.087	3.304	.094	.046	.140	.490
L32-19	.429	.429	.500	1.500	.500	2.000	.092	.031	.122	.333
Mean	.556	.623	.663	1.06	.48	1.54	.101	.061	.162	.76
S.D.	.194	.218	.242	.75	.26	.87	.033	.038	.041	.86

(continued)

Table 11 (cont'd)

Prob. No.	M11	M12	M13	M14	M15	M16	M17	M18	M19
L5-30	.472	.250	.444	3.722	3.472	2.167	.582	1.861	.900
L8-9	.257	.243	.162	3.216	2.892	2.595	.807	1.608	.925
L8-27	.434	.421	.263	4.526	3.658	2.684	.593	2.263	.950
L8-28	.559	.382	.294	5.412	4.647	3.147	.582	2.706	.850
L9-3	.688	.563	.250	7.125	5.313	4.063	.570	3.563	.800
L9-8	.559	.412	.500	5.382	4.353	4.000	.743	2.691	.850
L10-12	.222	.139	.278	2.944	2.306	1.972	.670	1.472	.923
L10-19	.271	.257	.229	3.286	2.771	2.143	.652	1.643	.921
L11-11	.170	.424	.303	7.667	5.909	3.970	.518	1.533	.868
L12-4	.223	.371	.343	7.771	6.829	4.686	.603	1.554	.921
L13-29	.371	.400	.400	6.600	4.429	2.686	.407	2.200	.921
L15-15	.465	.581	.323	11.194	7.097	3.774	.337	2.239	.886
L15-17	.168	.571	.321	13.321	10.286	8.464	.635	1.332	.800
L15-18	.222	.370	.333	9.074	7.111	4.778	.527	1.512	.771
L15-21	.141	.483	.207	15.000	11.759	9.897	.660	1.500	.829
L16-4	.297	.516	.387	9.484	7.387	5.774	.609	1.897	.886
L16-6	.178	.828	.207	22.621	14.345	8.034	.355	1.885	.829
L23-7	.208	.542	.167	14.208	11.708	6.250	.440	2.368	.750
L24-11	.317	.429	.619	10.571	8.238	5.857	.554	1.762	.656
L25-8	.460	.360	.320	5.000	3.640	1.720	.344	2.500	.781
L26-5	.276	.536	.571	10.071	8.036	6.500	.645	1.439	.875
L29-19	.216	.692	.269	14.192	7.423	3.692	.260	1.092	.839
L32-5	.673	.808	.692	23.346	14.692	5.077	.217	3.891	.867
L32-8	.661	.739	.565	23.522	15.739	10.522	.447	4.704	.767
L32-19	.667	.857	.214	16.357	12.000	9.000	.306	5.452	.483
Mean	.37	.487	.346	10.2	7.4	4.8	.523	2.27	.834
S.D.	.18	.189	.144	6.3	4.0	2.4	.155	1.08	.100

Commonly, the denominator of this ratio is taken to be the number of students who encountered the problem, so that a failure to respond is equivalent to an incorrect response; this definition is used whenever all students are expected to respond to every exercise presented to them. Since the AID course allowed students to omit problems without penalty, we chose the definition using as a divisor the number of students who actually attempted to solve the problem. The three measures of proportion correct are as follows.

M1: Proportion Correct on First Trial--(number of students who gave a correct solution on the first trial) ÷ (number of students who attempted the problem). M1 is taken as the primary measure of problem difficulty. This variable was discussed in Chapter IV. Its value ranges from 11.5% for Problem L29-19 to 85.7% for L10-19. The mean is 55.6% for the set of 25 problems.

M2: Proportion Correct on First Trial Disregarding Algebraic Errors--(number of students whose solution on first trial was correct except for possible algebraic errors) ÷ (number of students who attempted the problem). For several problems, e.g., L5-30, L9-8, L12-4, students used an algebraic formula that was incorrect, for instance, x*.3937 for x/.3937, although in all other respects the solution was correct. As pointed out in Chapter IV, disregarding errors in algebraic formulas increased the proportion correct substantially for some problems, with increases ranging up to nearly 100% (for L5-30). The mean of M2 is 62.3%, as compared to 55.6% for M1, and the standard deviation is 21.8%. Pursuing the comparison further, we find the correlation between M1 and M2 to be quite high

($r = .90$), as shown in the correlation matrix for the measures of problem difficulty (Table 12).

M3: Proportion Correct on All Trials--(number of students who achieved a correct solution on some trial) ÷ (number of students who attempted the problem). Had the teaching program consistently asked students to make another try if their first was not successful, M3 would be a better measure of problem difficulty. Since this was not done and since the amount of help offered varied considerably, this measure is not as satisfactory as either M1 or M2. The mean of M3 is higher than for either M1 or M2 (66.3% compared to 55.6% and 62.3%) and the standard deviation (24.2%) is also higher. M3 correlates slightly better with M2 than with M1 (.94 vs. .88) although M1, like M3, is based on completely correct solutions. This evidence in itself is not convincing but coupled with a closer study of the subsequent actions of students who made simple algebraic errors on their first try, it supports the inference that the teaching program is reasonably adept at detecting such errors, and offers effective assistance to the students who made them.

Number of Errors. The three measures of number of errors are all averages for the students who attempted the problem.

M4: Number of Syntax Errors per Student--(number of syntax errors) ÷ (number of students who attempted the problem). The average number of syntax errors ranges from .08 for Problem L10-12 to 2.88 for Problem L32-5, with a mean of 1.06 (S.D. = .75). One would expect errors to correlate negatively with proportion correct; this is true, and the correlation coefficients are all quite large: $r_{1,4} = -.72$, $r_{2,4} = -.83$, and $r_{3,4} = -.80$. The correlation with M2, which disregards algebraic

Table 12

Correlation Matrix for Measures of Problem Difficulty

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19
M1		.9036	.8801	-.7193	-.3195	-.7167	-.4407	.2452	-.1309	.3125	-.2612	-.6537	-.3735	-.6118	-.5197	-.2606	.6152	-.2651	.3958
M2			.9377	-.8303	-.2230	-.7839	-.4769	.5789	.1498	.5225	-.1754	-.7994	-.2397	-.7521	-.6675	-.4119	.6808	-.5116	.4418
M3				-.8030	-.2398	-.7802	-.4529	.4982	.0944	.4485	-.1118	-.7111	-.3833	-.7107	-.6170	-.3807	.6730	-.2234	.4339
M4					.3205	.9596	.5859	-.6268	-.1073	-.5320	.2204	.8649	.3251	.8635	.7661	.5074	-.7072	.3362	-.2708
M5						.5741	-.0508	.2852	.2217	.0647	.6002	.2807	.8584	.3955	.4091	.2847	-.1920	.5398	-.2352
M6							.4914	-.4571	-.0269	-.4406	.3687	.8309	.5360	.8639	.7838	.5233	-.6681	.4509	-.3041
M7								-.3410	.4916	-.5697	.2566	.3926	.1593	.1552	.0378	-.0759	-.3933	.0360	.0562
M8									.6509	.7439	.2886	-.6706	.2830	-.6470	-.6363	-.5634	.4172	-.0462	.2740
M9										.2285	.4736	-.3034	.3907	-.4736	-.5592	-.5836	.0696	-.0154	.3037
M10											-.0327	-.6248	.0431	-.4869	-.4904	-.4323	.3532	-.1605	.2596
M11												.3184	.3773	.1368	.0953	-.1352	-.2968	.8569	-.2802
M12													.1597	.8907	.8457	.5905	-.6840	.5573	-.5115
M13														.2524	.2333	.1776	-.1239	.2007	-.0130
M14															.9756	.7672	-.6082	.4339	-.4283
M15																.8449	-.4984	.4329	-.4891
M16																	-.0573	.1611	-.3468
M17																		-.4184	.3715
M18																			-.5628
M19																			

errors, is considerably higher than the correlation with M1, lending support to the notion that M2 may be a better measure of programming difficulty than M1.

M5: Number of Semantic Errors per Student--(number of semantic errors) $\div$ (number of students who attempted the problem). The average number of semantic errors (.48) is less than half the average number of syntax errors (1.06) and the range of values is also smaller (.11 to 1.15 with S.D. = .26). Furthermore, the correlation between syntax and semantic errors is quite low ($r_{4,5}$ = .32). The correlations between semantic errors and the three measures of proportion correct are also quite low, between -.22 and -.32, and semantic errors, unlike syntax errors, correlate better with M1 than with M2, as would be expected since a large number of semantic errors are taken out by M2.

M6: Number of all Errors per Student--M4 + M5. The average number of all errors is composed of about two-thirds syntax errors and one-third semantic errors, with a mean of 1.54 errors per problem. The correlation of M6 with syntax errors is remarkably high; $r_{4,6}$ = .96, as compared to $r_{5,6}$ = .57, the correlation with semantic errors. The correlations of M6 with the three measures of proportion correct follows the same pattern as M4, the number of syntax errors; again, there is a slightly higher correlation with M2 than with M1 (see Table 12).

Error Rates. Since the total number of errors may be dependent upon the number of commands given by the student, three measures of error rates were also defined, corresponding to the three measures M4, M5, and M6. For each of these, the number of errors is divided by the number of commands typed.

M7: Syntax Error Rate--(number of syntax errors) $\div$ (number of commands typed). The mean syntax error rate is 10% with a standard deviation of 3.3%. Recall that the count of syntax errors is a count of errors themselves and not a count of commands in error, so one cannot infer that an error rate of 10% indicates that one command in ten is in error but that, in ten commands, there is on the average one error. M7 corresponds to M4, the number of syntax errors, and the correlation is substantial but not spectacular ($r = .59$). In making comparisons with proportion correct, we find that M7 follows the same pattern as M4 but that the correlations are much lower; for example, $r_{1,7}$ is only -.44 whereas $r_{1,4}$ is -.72. As a prediction of proportion correct, the number of syntax errors would serve much better than the rate of syntax errors, accounting for 50% of the variance as opposed to 20%.

M8: Semantic Error Rate--(number of semantic errors) $\div$ (number of commands typed). As we would expect, the semantic error rate is lower than the syntax error rate (6% vs. 10%). Further comparing these two measures, we notice that the correlation is negative ($r_{1,8} = -.34$), and that the correlation between the semantic error rate and the number of syntax errors is also negative and has an even higher value ($r_{4,8} = -.63$). Furthermore, although all of the other error measures, M4 to M7, correlate negatively with proportion correct, as expected, the semantic error rate correlates positively with all three measures of proportion correct, and the values, though not high, are substantial (.25, .58, and .50). We do not know whether this phenomenon can be accounted for by characteristics that are peculiar to this set of problems or curriculum, or whether it is likely to be true for other programming problems given in other circumstances." The evidence here is strong enough to warrant a closer study of other data.

M9: Error Rate--(number of all errors) ÷ (number of commands typed). The mean of M9 is 16.2%, an average of one error for every six commands typed. M9 stands in the same relationship to M7 and M8 as M6 does to M4 and M5, and one would expect to find a similar pattern in the correlation matrix. The similarities are few, however, and one of the more noticeable variations is in the correlation with syntax errors. As we saw, M6, the number of errors, correlated extremely highly with M4, the number of syntax errors (r = .96). In comparison the correlation between M9 and M7 is only .49. The correlations with semantic errors are quite comparable: $r_{5,6}$ = .57 and $5_{8,9}$ = .65. Thus, the rate of all errors correlates better with the rate of semantic errors than with the rate of syntax errors, whereas the opposite is true if we measure the number of errors instead of the error rates. Although there was a fairly high correlation between M7 and M4, and a lower but not insignificant correlation between M8 and M5, the correlation between M9 and M6 is essentially nil (r = -.03). As a final comparison between number of errors and error rates, consider the value of M9 as a prediction of proportion correct: not more than 2% of the variance in proportion correct could be accounted for by the total error rate; on the other hand, M6, the total number of errors, could account for 50%.

From this discussion it is clear that measures of errors, and in particular the total error rate, measure problem difficulty along a different dimension than proportion correct. Although there is a very high correlation between the number of syntax errors and proportion correct, this may be because longer programs tend to be more difficult and also afford more opportunities for syntax errors; that this is not the whole story, however, is shown by the substantial correlation between syntax errors and proportion correct even after length is factored out ($r_{2,7} = -.48$, for example).

There are two more measures of error rates that may be of some interest:

M10: Ratio of Semantic Errors to Syntax Errors--(number of semantic errors) $\div$ (number of syntax errors). The mean of M10 is .76; on the average there are three semantic errors for every four syntax errors. The range of this variable is large, .04 to 4.33, with a standard deviation of .86. M10 is, as expected, negatively correlated with syntax errors, both M4 and M7, and positively correlated with M5 and M8, the two measures of semantic errors. Except for $r_{5,10}$, these correlations all have a magnitude of over .5, and the correlation with M8, the semantic error rate, is over .74. The correlations with proportions correct have the same pattern, and nearly the same values, as for M8.

M11: Ratio of Errors to the Length of the Expected Correct Solution--(number of errors) $\div$ [(number of commands in the expected correct solution) $\times$ (number of students who attempted the problem)]. This variable has a mean of .37 and its correlation with the other measures of error rates is fairly low except for the number of semantic errors ($r_{5,11} = .60$). M11 correlates negatively with proportion correct, as we expect of error measures, but the values are low.

The Number of Students Who Made Errors. For many problems, most of the errors were made by only a few students. The fact that students skipped different problems may cause some doubt about the reliability of measures M4 to M11. The last two error measures do not share this defect.

M12: Number of Students Who Made Syntax Errors--(number of students who made syntax errors) ÷ (number of students who attempted the problem). On the average 48.7% of the students made one or more syntax errors, and the range is from 13.9% for Problem L10-12 to 85.7% for L32-19. The correlation with the average number of syntax errors is quite high ($r_{4,12}$ = .86) but not nearly so high with the syntax error rate ($r_{7,12}$ = .39). Again we see evidence of the difference between syntax and semantic errors: the correlation between the semantic error rate and number of students who made syntax errors is negative and quite high ($r_{8,12}$ = -.67). The correlations of M12 with the three measures of proportion correct follow the same pattern as for the other syntax error measures (M4 and M7), showing the greatest correlation with M2, the proportion correct discounting algebraic errors. These three correlations are not quite as high as for M4 but are considerably higher than for M7.

M13: Number of Students Who Made Semantic Errors--(number of students who made semantic errors) ÷ (number of students who attempted the problem). On the average 34.6% of the students made one or more semantic errors. The low value of 16.2% is for Problem L8-9 and the high of 69.2% is for L32-5. Again the correlation is much higher with the number of errors than with the error rate: $r_{5,13}$ = .86, whereas $r_{8,13}$ = .28. As we have come to expect, there is little correlation with measures of syntax errors: $r_{4,13}$ = .32, $r_{7,13}$ = .16, and $r_{12,13}$ = .16. The lowest of all correlations for M13 is $r_{10,13}$ = .04, the correlation between M13 and the ratio of semantic to syntax errors. The correlations of M13 with the measures of proportion correct are negative but the values are not high (less than .4).

If M12 and M13 were to be taken as replacements for M4 and M5, we can see that not much would be gained; the correlations $r_{4,12}$ and $r_{5,13}$ are both quite high.

Leaving measures of errors, we turn now to measures of effort.

<u>Effort Expended</u>. Five measures of effort are defined, the last two of which are ratios.

M14: Number of Commands Typed--(number of commands typed) ÷ (number of students who attempted the problem). The number of commands typed has a wide range, from 2.9 for Problem L10-12 to 23.5 for L32-8. The mean of this variable is 10.2. We would expect this variable to vary directly with problem difficulty and hence inversely with proportion correct; this expectation is borne out and the correlations with M1, M2, and M3 are quite high (e.g., $r_{2,14} = -.75$). Looking at the correlation with M4, we confirm the suspicion that the number of commands and the number of syntax errors are statistically dependent ($r = .86$), and the correlation with syntax error rate is correspondingly satisfyingly low ($r_{7,14} = .16$). Still looking at the correlation vector for M14, we find that the correlation with the semantic error rate is fairly high but is negative ($r_{8,14} = -.65$).

M15: Number of Commands Executed--(number of commands executed) ÷ (number of students who attempted the problem). The average number of commands executed is 7.4, about three-fourths of the commands typed, and the correlation with commands typed is extremely high ($r_{14,15} = .98$). The correlation vector for M15 is quite similar to that for commands typed, including the high negative correlation with the semantic error rate ($r_{8,15} = -.64$).

M16: Commands Used in Partially Correct Solutions--(number of commands used in correct or partially correct solutions) ÷ (number of students who attempted the problem). The average number of commands used in partially correct solutions is 4.8, as compared to 7.4 executed commands and 10.2 typed commands. As mentioned in Chapter IV, fewer than half the typed commands contributed toward a correct solution. The correlations of M16 with both commands typed and commands executed are quite high: $r_{14,16} = .77$ and $r_{15,16} = .85$. The correlation vector for M16 again follows the pattern set by M14 and M15 although the magnitudes are generally lower. Again we note that the correlation with the syntax error rate is low (-.08), and that the correlation with the semantic error rate is still negative, though somewhat less than for M14 and M15 (-.56 as compared to -.65 and -.64). Since it is hard to believe that, as a general rule, the rate of semantic errors declines with the number of commands, it seems likely that this result is caused by unidentified peculiarities of the set of problems or the curriculum.

M17: Ratio of Commands Used in Partially Correct Solutions to Commands Typed--(number of commands used in correct or partially correct solutions) ÷ (number of commands typed). This variable measures the proportion of useful effort; the mean is 52.3% and the range is from 21.7% to 80.7%. In examining the correlation vector for M17, we see that M17 correlates well with the three measures of proportion correct ($r > .6$) and in the expected direction. As expected, it correlates negatively with the number of errors, although the correlation with the number of semantic errors is not high, -.19 as compared to -.71 for syntax errors. M17 also correlates negatively with the syntax error rate ($r = -.39$), but the correlation with the semantic error rate is positive ($r = .42$), and there is a very low correlation with the total error rate ($r = .07$).

M18: Commands Typed ÷ Length of Expected Correct Solution--(number of commands typed) ÷ [(number of commands in the expected correct solution) × (number of students who attempted the problem)]. This last measure of effort is akin to an efficiency measure: it measures the amount of effort in terms of a standard, and presumably relatively efficient, solution. The range of M18 is from 1.09 to 5.45 with a mean of 2.27; on the average, students did over twice as much as was needed to achieve a correct solution. In the correlation vector for M18 we find only one sizable value: $r_{11,18} = .86$. Since M11 is also a ratio with the length of the expected solution in the denominator, this value is a reflection of the high correlation between the total number of errors and the number of commands typed. Of some interest are the very low correlations with M7, M8, and M9, the three measures of error rates ($|r| < .05$); relative efficiency, as measured by M18, seems to have little statistical relation to error rates.

Number of Students Who Attempted Problem. This final measure of problem difficulty might have been classified as another measure of effort for it simply measures the proportion of students who made some effort to solve the problem.

M19: Students Who Attempted Problem--(number of students who attempted problem) ÷ [(number of students who attempted the problem) +

(number of students who skipped the problem)]. Although there are 40 students represented in the data, the denominator of M19 is not always 40 since some students did not progress far enough through the course to encounter all of the 25 problems. The values of M19 range from 48.3% to 95% with a mean of 83.4%. We do not know why some students skip certain problems, whether it is because a certain problem is perceived as difficult, or perhaps as too easy and hence a waste of time. An examination of student protocols indicates that there was neither a small group of students who consistently skipped problems, nor a particular problem or set of problems singled out. In fact, only two problems were skipped by more than one quarter of the students. A study of the correlation vector for M19 does not shed much more light on this question. There are two values greater than .5. The value of $r_{12,19}$ is -.51, indicating some statistical relationship with M12, the number of students who made syntax errors. The value of $r_{18,19}$ is -.56, indicating a relationship with relative efficiency. At the other end of the scale, we see very low correlations with the syntax error rate (r = .06) and with the number of students who made semantic errors (r = -.01).

Several facts emerge from the above discussion of the 19 variables related to problem difficulty. Foremost is that many of these measures are statistically unrelated to one another. If all of them are measuring some aspect of problem difficulty, then it is clear that the measurements are along several, quite different dimensions. There are, of course, strong similarities between certain pairs of measures. M1 and M2, for example, which are both measures of proportion correct, are closely related both conceptually and statistically. For the most part, those pairs that one would expect to be closely related do correlate highly and in the expected direction. There is a striking exception to this in the set of measures based on semantic errors; these measures, in particular M8 and M10, do not relate to measures of proportion correct or to measures of syntax errors in the way one would expect. In fact, if we had looked only at M2 as a measure of proportion correct and M10 as a measure of the errors, we might have been tempted to conclude that the error rate is highest for the easiest problems. We draw no such conclusions, however, as we have not been able to formulate any intuitively satisfying hypothesis that could account for this apparent anomaly in the data.

Having 19 measures of problem difficulty is an embarrassment of riches, and for more detailed study we chose from among them a smaller, more manageable subset. As mentioned before, we consider M1 to be the primary measure of problem difficulty because it is the most similar to measures of problem difficulty used by other researchers and our results can thus be more readily compared to results obtained by others. For reasons already mentioned, we feel that M2, the proportion correct disregarding errors in algebraic formulas, is a more satisfactory measure of programming difficulty per se. M1 and M2 seem to measure very similar aspects of problem difficulty, so for variety we also chose four other measures that seemed to be quite unrelated to M1 and M2 and to one another; one is a measure of syntax errors (M7), another a measure of semantic errors (M13), the third is a measure of efficiency or effort (M18), and the last is the number of students who attempted the problem (M19). For ease of reference these selected measures are listed in Table 13, which also shows the correlations between each pair.

In pursuing the study of these six measures, we are interested in discovering what characteristics of the problems or of the curriculum influence problem difficulty, and how well we could have predicted problem difficulty from an a priori evaluation of these characteristics. The tool we used in this study was step-wise multiple linear regression using ten independent variables to predict the values of the six selected measures of problem difficulty. The ten variables are defined independently of the data; some of them measure characteristics of the problems themselves (ARG, FCT and INPUT), some measure aspects of the curriculum context (LES, HELP, VOCAB, and NEW), and some are obtained from the expected correct solutions and are hence dependent upon both the problems and their context (IF and LNG). The ten variables are described below and their values for each problem are given in Table 14.

IF: The variable IF is the proportion of conditional commands (i.e., the commands that contain an IF clause) used in the expected correct responses listed in Chapter II. The values of IF vary from 0% to 67% with a mean of 14% and a standard deviation of 20%, as shown in Table 14.

ARG: This variable depends upon the mathematical function required by the problem. The values of ARG are

0	if there is no argument for the function
1	if there is one real argument
2	if there are two real arguments
3	if the argument is a stored list

The mean of ARG is 1.6.

Table 13

Six Selected Measures of Problem Difficulty

Description of Six Measures	M1	M2	M7	M13	M18	M19
M1: Proportion Correct. (number of students who gave a correct solution on their first trial) ÷ (number of students who attempted problem)	1.00	.90	-.44	-.37	-.27	.39
M2: Proportion Correct up to Algebra. (number of students whose solutions on first trial were correct except for possible errors in algebraic formulas) ÷ (number of students who attempted problem)		1.00	-.48	-.24	-.31	.44
M7: Syntax Error Rate. (number of syntax errors) ÷ (number of commands typed)			1.00	.16	.04	.06
M13: Number of Students who Made Semantic Errors. (number of students who made semantic errors) ÷ (number of students who attempted the problem)				1.00	.20	-.01
M18: Efficiency. (number of commands typed) ÷ [(number of commands in the expected correct solution) × (number of students who attempted the problem)					1.00	-.56
M19: Students who Attempted Problem. (number of students who attempted problem) ÷ [(number of students who attempted problem) + (number of students who skipped problem)]						1.00

Table 14

Values of Independent (Problem) Variables

Problem Number	IF	ARG	FCT	REIT	LNG	INPT	LES	HELP	VOC	NEW
L5-30	.00	1	1	0	2	3	5	0	2	1
L8-9	.00	1	1	0	2	3	8	2	3	1
L8-27	.00	2	1	0	2	2	8	0	3	0
L8-28	.00	1	1	0	2	5	8	0	3	0
L9-3	.00	2	1	0	2	4	9	0	3	1
L9-8	.00	2	1	0	2	4	9	0	3	1
L10-12	.00	1	1	0	2	4	10	1	5	1
L10-19	.00	1	1	0	2	10	10	1	5	0
L11-11	.00	1	3	0	5	5	11	0	6	1
L12-4	.00	2	1	0	5	1	12	1	7	1
L13-29	.00	1	1	0	3	4	13	0	7	0
L15-15	.40	2	1	0	5	0	15	1	8	1
L15-17	.30	2	1	0	10	0	15	1	8	0
L15-18	.50	2	1	0	6	0	15	1	8	0
L15-21	.30	2	1	0	10	0	15	0	9	1
L16-4	.20	1	1	0	5	0	16	2	10	1
L16-6	.67	2	1	0	12	9	16	0	10	0
L23-7	.17	0	1	1	6	0	23	0	12	0
L24-11	.17	1	1	1	6	0	24	0	12	0
L25-8	.00	1	1	0	2	7	25	2	12	0
L26-5	.00	1	2	1	7	0	26	1	12	0
L29-19	.46	2	1	0	13	1	29	0	12	0
L32-5	.00	3	2	1	6	2	32	0	13	1
L32-8	.00	3	1	1	5	2	32	2	13	0
L32-19	.33	3	1	1	3	1	32	0	13	0
Mean	.14	1.60	1.16	.24	5.00	2.68	16.7	.60	7.96	.44
S.D.	.20	.76	.47	.44	3.29	2.87	8.6	.76	3.81	.51

FCT: Most programs written for the 25 problems in this set define only a single mathematical function but a few define several functions (of the same number of arguments). The value of FCT is the number of mathematical functions defined, and may be 1, 2, or 3.

REIT: This is a 0-1 "reiteration" variable whose value is 0 if no loops or subroutines are required and 1 otherwise.

LNG: This variable is the number of commands in the expected correct solution. The values range from 2 to 13 with a mean of 5.

INPT: The number of sets of input values specified in the problem is given by INPT, whose value ranges from 0 to 10 with a mean of 2.7.

LES: This variable, the lesson number, is a measure of the position of the problem in the curriculum.

HELP: This variable measures the amount of help given in the problem statement. Problem statements occasionally include an example of a closely related program, or part of such a program, and HELP is a measure of this kind of assistance. HELP = 0 if no model was given, HELP = 1 if a partial model was given, and HELP = 2 if a complete model was given. The value of HELP is non-zero for 11 of the 25 problems, with a mean of .6.

VOC: This variable measures the amount of AID vocabulary that had been presented by the curriculum before the problem was given. The lexical items that are counted are

TYPE
SET
LET
DO STEP (used directly)
FOR

DO PART (used directly)
DEMAND
IF
AND
TO
DO (used indirectly)
FORM
Indexed variables

NEW: This is a 0-1 variable that depends upon whether the problem requires the use of a command or function that has not been used in a preceding programming problem. For this definition "programming problems" are taken to be any exercises that require the use of the AID interpreter other than those problems that require only that the student copy verbatim AID commands printed by the teaching program. The "new" commands and functions considered here are not restricted to the list given above for VOC. Approximately half of the problems do require the use of a new word, and hence have a value of 1 for NEW.

The ten variables described above were used as independent variables in step-wise multiple linear regressions in an attempt to discover which were effective in accounting for problem difficulty. For this purpose it is best to use variables that are statistically independent of one another. This goal is difficult to achieve, and was approached with only moderate success by this set of variables, as can be seen from the correlation matrix given in Table 15. There are five pairs of independent variables for which the correlation is greater than .5. The first of these is IF-LNG. In a preliminary study, the variable IF was defined to be the number of conditional commands used in the expected correct solution, rather than the proportion of conditional commands.

Table 15

Correlation Matrix for Independent (Problem) Variables

	IF	ARG	FCT	REIT	LNG	INPT	LES	HELP	VOC	NEW
IF	1.000	0.277	-0.246	-0.081	0.728	-0.185	0.238	-0.136	0.383	-0.262
ARG		1.000	-0.046	0.175	0.232	-0.175	0.390	-0.071	0.252	0.043
FCT			1.000	0.210	0.080	0.039	0.135	-0.162	0.119	0.216
REIT				1.000	0.087	-0.369	0.768	-0.075	0.683	-0.309
LNG					1.000	-0.278	0.406	-0.166	0.541	-0.225
INPT						1.000	-0.316	-0.023	-0.333	-0.100
LES							1.000	0.091	0.941	-0.403
HELP								1.000	0.137	0.043
VOC									1.000	-0.400
NEW										1.000

When it was found that this variable correlated highly with length (r = .85), an attempt was made to reduce the correlation by dividing by length. Some reduction was accomplished but it was not as great as was hoped for (from .85 to .73). A study of the values of IF and LNG (see Table 14) reveals that for 9 of the 25 problems the expected correct solution contains only two commands, and that for these problems the value of IF is zero; this fact alone explains the high value of r and is sufficient reason for considering separate analyses of programs with and without conditional commands. This was not done here because the size of the sample is too small to warrant subdivision.

The second pair of highly correlated variables is REIT-LES, for which r = .77. Whether or not there are loops or subroutines (REIT) is highly dependent upon lesson number (LES). For the first 17 problems, included in Lessons 1 to 16, the value of REIT is zero. The value of REIT is 1 for six of the remaining eight problems. No reasonable way of transforming either REIT or LES to reduce the correlation was apparent.

VOC is also highly correlated with LES (r = .94) and with REIT (r = .68). It is to be expected that the amount of vocabulary introduced will be dependent upon lesson number and we would expect VOC and LES to account for approximately the same variance in problem difficulty.

There is one remaining pair of variables with a high correlation: for LNG-VOC the value of r is .54. In Chapter III we observed a similar phenomenon; that the total number of commands in the data tended to increase with problem number but that the increase depended more upon an increase in the variety of commands than upon an increase in the occurrence of a given kind of command. That comment referred to the data, whereas LNG is a function of the expected correct response; the same observation seems to be true for both, however.

As a preliminary view of the relationships between the ten independent variables and the six selected measures of problem difficulty, a correlation matrix is shown in Table 16. The correlation vectors for M1 and M2, the two measures of proportion correct, are strikingly similar. Both M1 and M2 are correlated highly (negatively) with LES; there are also substantial correlations with IF, ARG, LNG, and VOC; and the correlations with FCT and INPT are quite low. The pattern of the correlation vector for M19 shows some resemblance to the vectors M1 and M2 although the similarities are not as great as between M1 and M2. There are few similarities between the other correlation vectors, and the difference between M7, the syntax error rate, and M13, the number of students who made semantic errors, is marked. REIT, for example, correlates quite well with M13 ($r = .5$) but not at all with M7 ($r = .025$). Also, the coefficient for IF-M7 is positive whereas it is negative for IF-M13. In general, the correlations with M2 are high, followed closely by M1, and the correlations for M7 are low.

Using BMD02R we ran six step-wise regressions, one for each of the six selected measures of problem difficulty, and derived linear equations for the prediction of each of those measures. These equations (with coefficients rounded) are given in Table 17. For ease of reading we have transformed each equation to yield percentages rather than fractions. Our primary purpose in using step-wise regressions was not to produce these linear models, however, but to determine which of the independent variables had the greatest influence and to find out how

Table 16

Correlations Between Independent Variables and Six Measures of Problem Difficulty

Independent Variable	Measure of Problem Difficulty					
	M1	M2	M7	M13	M18	M19
IF	-0.494	-0.579	0.277	-0.308	-0.151	-0.358
ARG	-0.439	-0.506	0.265	0.225	0.572	-0.299
FCT	-0.034	-0.129	-0.014	0.295	-0.055	0.125
REIT	-0.387	-0.443	-0.025	0.496	0.531	-0.576
LNG	-0.491	-0.603	0.256	-0.022	-0.324	-0.133
INPT	0.198	0.238	-0.215	-0.245	0.002	0.248
LES	-0.529	-0.648	0.245	0.382	0.464	-0.599
HELP	0.475	0.394	-0.064	0.026	-0.070	0.166
VOC	-0.436	-0.585	0.155	0.303	0.270	-0.555
NEW	0.252	0.358	-0.162	0.045	-0.085	0.395

Table 17

Linear Models for the Prediction of Problem Difficulty

$$\begin{aligned}
M1 \times 100 &= 69 - 50\ IF + 2\ ARG + 1\ FCT - 5\ REIT - 1\ LNG - 0.3\ INPT - 4\ LES \\
&\quad + 9\ HELP + 7\ VOC - 4\ NEW \\
M2 \times 100 &= 91 - 56\ IF - 2\ ARG - 7\ FCT - 4\ REIT - 1\ LNG - 3\ LES \\
&\quad + 7\ HELP + 5\ VOC + 3\ NEW \\
M7 \times 100 &= 10 + 5\ IF - 1\ ARG + 1\ FCT - 5\ REIT - 0.3\ INPT + 1\ LES \\
&\quad - 0.1\ HELP - 1\ VOC - 1\ NEW \\
M13 \times 100 &= 20 - 41\ IF + 6\ ARG + 3\ FCT + 10\ REIT + 1\ LNG - 1\ INPT - 1\ LES \\
&\quad + 2\ VOC \\
M18 \times 100 &= 131 + 76\ IF + 72\ ARG - 25\ FCT + 82\ REIT - 22\ LNG + 7\ INPT + 4\ LES \\
&\quad - 22\ HELP + 2\ VOC + 17\ NEW \\
M19 \times 100 &= 85 - 31\ IF - 1\ ARG + 1\ FCT - 7\ REIT + 2\ LNG + 0.2\ INPT - 0.4\ LES \\
&\quad + 3\ HELP - 0.3\ VOC + 2\ NEW
\end{aligned}$$

much of the variance in problem difficulty could be accounted for by linear combinations of the ten independent variables.

Table 18 is a summary table of the dynamics of the regression, showing for each measure of problem difficulty the order in which the independent variables entered into the regression and the amount of variance accounted for at each step. The total amount of variance accounted for is shown at the bottom of each column, and we will start with those totals. The first fact of interest is that over 80% of the variance of M2 and M18 are accounted for; that is, the models serve quite well for predicting the proportion correct up to algebra and the amount of effort made by students relative to a fixed set of correct solutions. The models for M1 and M19 are also reasonably good; 76% of the variance in proportion correct can be accounted for, and 67% of the variance in the number of students who attempt the problem. The models for M7, syntax error rate, and M13, the number of students who made semantic errors, are less satisfactory, with less than 50% of the variance accounted for.

In considering these figures it should be kept in mind that we are using ten independent variables to account for the variance in 25 problems, and thus would expect to account for a substantial portion of the variance even if our independent variables were poorly chosen. For a comparison, let us see what the results would be if we selected only five of the ten independent variables (the best five in each case). The amount of variance accounted for by the first five variables to enter the regression is also shown at the bottom of Table 18. For all but one regression, the first five variables account for 95% to 98% of the

Table 18

Amount of Variance in Problem Difficulty Accounted for by Linear Combinations of Ten Independent Variables

	Dependent Variable											
	M1 Proportion correct		M2 Proportion correct up to algebra		M7 Syntax error rate		M13 Number of students who made semantic errors		M18 Ratio of commands typed to number of commands in expected correct solution		M19 Proportion of students who attempted problem	
Step number	Vari-able	R^2	Vari-able	R^2	Vari-able	R^2	Vari-able	R^2	Vari-able	R^2	Vari-able	R^2
1	LES	.280	LES	.420	IF	.077	REIT	.246	ARG	.327	LES	.358
2	HELP	.556	HELP	.627	ARG	.115	IF	.319	LNG	.548	IF	.408
3	IF	.645	IF	.763	INPT	.135	ARG	.372	REIT	.760	LNG	.565
4	VOC	.738	VOC	.817	NEW	.155	LNG	.411	INPT	.785	REIT	.630
5	LNG	.744	FCT	.836	REIT	.193	INPT	.420	IF	.805	HELP	.659
6	NEW	.747	LNG	.842	LES	.252	FCT	.427	LES	.817	FCT	.664
7	ARG	.752	NEW	.844	VOC	.397	VOC	.433	HELP	.831	NEW	.667
8	REIT	.754	REIT	.845	FCT	.416	LES	.441	FCT	.836	INPT	.669
9	INPT	.755	ARG	.847	HELP	.417	*		NEW	.839	ARG	.670
10	FCT	.756	*		*		*		VOC	.839	VOC	.670
Total variance accounted for	75.6%		84.7%		41.7%		44.1%		83.9%		67.0%	
Variance accounted for by first five variables	74.4%		83.6%		19.3%		42.0%		80.5%		65.9%	

*Remaining variables did not enter into regression.

variance accounted for by the full set of ten variables; in other words, our prediction would be essentially as good if we used only one-half of the independent variables. The noticeable exception to this rule is the model for the prediction of M7, the syntax error rate. For this measure of problem difficulty the prediction with ten variables is not good (only 42%) and the prediction based on the best five variables is quite unacceptable (19%). In short, our derived linear model for the prediction of M7 is worthless for practical purposes.

For the cases in which we can account for a reasonable amount of the variance, it is instructive to look more closely at the order in which the independent variables enter into the regression. For three regressions, LES is the first variable, from which we can conclude that the position in the curriculum is an extremely influential factor in problem difficulty. On the average LES alone accounts for more variance than any other single variable. The second most influential variable seems to be IF, which is among the first five variables to enter the regression in all cases. REIT is among the first five variables in four out of the six cases, and would have appeared more influential if LES, with which it is highly correlated, had been removed from the list. Another variable of some importance is HELP which entered second in two cases and fifth in one case. In summary, we conclude that in predicting problem difficulty the variables with greatest influence are the position in the curriculum, the predicted proportion of conditional commands, whether or not loops or subroutines are required, and whether or not the curriculum offers an example for the student to model his solution on.

The first and last of these might be characterized as curriculum-dependent, whereas the other two are more related to the problem itself than to the context in which it is found.

These conclusions are subject to some interpretation, however, For example, in the regressions M1 or M2 (proportions correct), REIT entered in only the eighth step, and we might conclude that whether or not loops or subroutines are required has little effect on proportion correct. This conclusion cannot be drawn with impunity, however, because of the high correlations between REIT and both LES and VOC. Since both LES and VOC entered into the regressions earlier, they took out a great deal of the variance that would otherwise be attributed to REIT.

Another fact worth commenting on is that although LES, HELP, and IF entered as the first three variables in the regressions for M1 and M2, they did not enter until the fifth, sixth, and seventh steps in the regression for M18, the relative efficiency of students' work.

Before turning our attention to other aspects of students' performance on programming problems, we would like to make a few comments on the analysis described in this chapter. First, although we defined and explored in some depth a large number of measures of different aspects of problem difficulty, there is another sizable set of variables that might be even more precise measures of problem difficulty, and those are measures based on the time required by students to produce solutions. We did not consider time-dependent measures here because the instructional system did not record elapsed time in any precise way. The reader who is interested in analyses of problem solving behavior using time-dependent measures of problem difficulty is referred to Dr. James Maloney's paper

"An Investigation of College Student Performance on a Logic Curriculum in a Computer-assisted Instruction Setting." The methods of analysis used by Dr. Maloney are similar to those used in this chapter, and, in fact, provided a model from which this author drew ideas about both method and definitions of independent variables.

One independent variable of possible importance was inadvertently omitted, a measure of the amount of guidance available to the students in the optional hints. This variable is akin to the HELP variable used to measure the (non-optional) guidance given in the problem statement. In view of the fact that HELP was quite effective in predicting M1 and M2, it seems reasonable that a HINT variable might also have been worth considering.

CHAPTER VII

Classification of Correct and Nearly Correct Solutions

In the preceding chapters we discussed the number and distribution of correct solutions. In this chapter and the next we study the kinds of correct solutions. Correct, and nearly correct, solutions are classified by type using four different methods of classification, two based on the forms of programs and two based on the functions. As noted in Chapter IV, 427 of the 747 first attempts made by students were correct. In addition, there were 124 solutions nearly enough correct that they could be unambiguously classified according to each of the four classification schemes. These nearly correct solutions are included in the analyses described here, giving a total of 551 student-written programs, an average of 22 per problem.

In this chapter we use "solution," or more loosely "program," to refer to both the stored program and the direct commands used to execute it. For the first few problems, through L11-11, a solution is not considered correct unless the student executed the program using the input values specified in the problem statement. (He could, of course, use additional values also.) After L11-11 a correct solution must include the commands needed to execute the program but the actual values used are immaterial. Because of this discontinuity in the grading scheme our definitions of program equivalence will contain special clauses for the treatment of solutions written after Problem L11-11.

For the first two definitions of program equivalence we are concerned with the forms of programs, and will define equivalence in terms of substitution of equivalent commands or sequences of commands. These first two kinds of equivalence, which we will call formal identity and formal equivalence, are defined in terms of allowable substitutions as follows: two programs are said to be formally identical (formally equivalent) if one of them can be transformed into the other by any finite sequence of substitutions, with possible repetition, from the list of substitutions allowable for formal identity (formal equivalence).

Since the allowable substitutions for formal equivalence include the substitutions allowed for formal identity, it follows that any two formally identical programs are also formally equivalent, although the converse is not necessarily true. Furthermore, all of the allowable substitutions preserve semantics, so any two that are either formally identical or formally equivalent will also be functionally equivalent (which will be our fourth definition of program equivalence).

The formal substitution rules are described below. Rules 1 to 7 define formal identity, and Rules 1 to 15 define formal equivalence.

Rule 1. If two commands are identical except for optional spaces, one may be substituted for the other. For example, spaces may be freely used in simple algebraic expressions, so these two commands are equivalent under Rule 1:

```
TYPE X + Y - Z
TYPE X+Y-Z
```

Rule 2. If two literal numerals are equal when rounded to three decimal places, one may be substituted for the other. The following are equivalent under Rule 2:

```
0.3937
 .3937
 .394
```

Rule 3. If two programs differ only in literal step numbers, one may be substituted for the other. The step numbers must be in the same numerical sequence within parts, and references to the steps (as in DO or TO commands) are substituted for concurrently. As an example, the following two programs are equivalent under Rule 3:

```
1.1 DO PART 2 IF X < 1          3.15 DO PART 1 IF X < 1
1.2 TYPE X                      3.17 TYPE X
2.1 SET X = - X                 1.3 SET X = - X
DO PART 1                       DO PART 3
```

Notice that although the sequence of steps within a part must remain in numerical order, part numbers need not.

Rule 4. If two LET commands differ only in the letter used as a dummy variable, one may be substituted for the other. Rule 4 applies only to variables bound within a LET command whereas Rule 5 applies to other variables as well.

Rule 5. If two programs differ only in the letters used for variables, one may be substituted for the other. The variables referred to here may be real variables, names for functions, or names for lists of numbers.

The next two rules are applicable only for problems after L11-11.

Rule 6. In a direct command of the form

DO PART n FOR x = m_1

m_1 may be replaced by m_2 where m_1 and m_2 are any numbers, list of numbers, or range specification. In the above n is a part number and x is any variable.

Rule 7. In a direct command of the form

SET x = n_1

n_1 may be replaced by n_2 where n_1 and n_2 are any numbers.

These first seven rules define formal identity. Although the above definitions are not stated with complete precision, they can be reformulated precisely to define a decision procedure for formal identity, the only one of our four equivalence relations that does admit such a procedure. When the set of solutions for each problem was classified using the above rules, the most numerous class was labeled I_1, the second most numerous I_2, etc. These designations are used in Appendix A which contains a complete list of the types of solutions for each of the 25 problems. For some problems a great reduction in the number of types is attained by this method of classification. For example, for Problem L8-9, there were 36 correct solutions but only four distinct types after reduction by formal identity. For other problems, the differences between students' programs are less trivial and consequently the reduction by formal identity is less effective. For instance, no two solutions for L16-6 are formally identical, so no reduction in the number of types is achieved. The number of types, or equivalence classes, under formal identity varies from 3 to 13 as shown in Table 19. There are an average of 8 equivalence classes per problem with an average of 2.7 solutions per equivalence class. As we will see, formal identity is the weakest of the four methods of classification used.

Table 19

Number of Programs in Each Equivalence Class, Using Four Definitions of Program Equivalence

Problem Number	Number of Programs	Number of Equivalence Classes when Partitioned by...			
		Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L5-30	33	7	5	5	1
L8-9	36	4	1	1	1
L8-27	29	12	5	4	1
L8-28	26	12	2	2	1
L9-3	23	9	3	3	1
L9-8	26	6	4	4	2
L10-12	27	10	2	1	1
L10-19	30	3	1	1	1
L11-11	26	13	5	6	1
L12-4	34	3	3	2	1
L13-29	25	8	7	5	2
L15-15	20	14	11	2	2
L15-17	23	7	6	3	1
L15-18	18	10	6	3	2
L15-21	25	6	3	1	1
L16-4	29	12	7	4	2
L16-6	8	8	8	5	2
L23-7	13	11	11	7	3
L24-11	18	6	6	3	1
L25-8	23	4	1	1	1
L26-5	22	12	8	6	2
L29-19	6	5	5	4	1
L32-5	9	8	7	8	5
L32-8	16	9	8	6	3
L32-19	6	5	4	4	3
Totals	551	204	129	91	42

Formal equivalence, the second equivalence relation, is also defined in terms of substitution rules. In addition to the above seven rules, Rules 8 to 15 are used to determine formal equivalence.

Rule 8. The phrases

FOR x = a(b)c

and

FOR x = a,a+b,a+2b,...,c

may be interchanged provided the allowable length for AID commands is not exceeded. In the above, x refers to any variable; a, b, and c are real numbers; and a+b,a+2b, etc., are real numbers whose values are a+b, etc. Under this rule the following phrases are equivalent:

FOR A = 4(2)9

FOR A = 4,6,8,9

The next five rules provide for substitutions of single commands for sets of commands or for permutations in the sequence of commands provided such substitutions do not change the function of the program. To avoid semantic changes, we require that the commands to be substituted for be contained between "critical points" in the program. A critical point is either the beginning or end of a part or a step to which branching may occur. Thus, if Rule 11, for example, would ordinarily allow us to interchange Steps 7.3 and 7.4, this would be allowed only if there is no branch command (TO or DO) elsewhere that refers to Step 7.4. This restriction applies to Rules 9 to 13.

Rule 9. The sequence of commands

```
TYPE e1
TYPE e2
  :
TYPE en
```

may be interchanged with the single command

```
TYPE e1,e2,...,en
```

where e_i's are algebraic expressions, provided that the single TYPE command does not exceed the allowable length for AID commands.

<u>Rule 10</u>. The sequence of n commands

```
DO e
DO e
 :
DO e
```

may be interchanged with the single command

```
DO e, n TIMES
```

where e is the specification of a part or step.

<u>Rule 11</u>. The two commands

```
SET x = e
DEMAND y
```

may be interchanged if the expression e contains no occurrences of the variable y and if the variables x and y are not identical. Thus, we can interchange

```
SET A = 2*B
DEMAND C
```

or

```
SET A = 2*A
DEMAND C
```

but not

```
        SET A = 2*C
        DEMAND C
```

or

```
        SET M = 2*A
        DEMAND M
```

Rule 12. The two commands

```
        SET x = e1
        TYPE e2
```

may be interchanged if the expression e_2 contains no occurrences of the variable x.

Rule 13. The two commands

```
        SET x = e1
        SET y = e2
```

may be interchanged if x and y are distinct variables, and e_2 contains no occurrence of x, and e_1 contains no occurrence of y. Either or both of the SET commands may have appended IF clauses, provided x and y do not occur in the Boolean expressions used in the IF clauses.

By a suitable reformulation of Rules 8 to 13, a decision procedure could be written for an equivalence relation determined by them. The next two rules for formal equivalence do not admit of a decision procedure, however, since both are based on algebraic equivalence.

Rule 14. If a command contains an algebraic expression e_1, then any algebraically equivalent expression e_2 may be substituted for it.

Rule 15. If a command contains a Boolean expression e_1, then any logically equivalent expression e_2 may be substituted for it. As an example, the two commands

```
TYPE.X IF X < Y + 7
```

and

```
TYPE X IF X - 7 < Y
```

are equivalent under Rule 15. Notice, however, that the commands

```
TYPE X IF X < Y + 7
```

and

```
TYPE X IF X < 7 + Y
```

are equivalent under either Rule 14 or Rule 15. Thus, Rules 14 and 15, unlike other pairs of rules, are not independent.

The 15 rules above constitute the complete definition of formal equivalence. As mentioned, only the last two rules prevent the formulation of a decision procedure for formal equivalence. It is clear from an inspection of the solutions listed in Appendix A that a few simple rules for algebraic substitution would serve to define a decision procedure for the algebraic expressions found in the data, so a partial solution to this problem could be attained if it were desirable to implement a routine for determining formal equivalence.

Under formal equivalence a greater reduction in types is made than under formal identity, as can be seen from Table 19. The 551 solutions reduce to 129 types, an average of five equivalence classes per problem as compared to the eight equivalence classes per problem for formal identity. Under formal equivalence there is an average of 4.3 solutions per equivalence class as compared to 2.7 under formal identity.

We remark again that formal identity is included in formal equivalence so that any two programs that are formally identical are also formally equivalent, and any two programs that are not formally equivalent are also not formally identical. The next two equivalence relations to be discussed, <u>algorithmic equivalence</u> and <u>functional equivalence</u>, also include formal identity. Functional equivalence also includes formal equivalence but it is not the case that algorithmic equivalence includes formal equivalence. Thus, it is possible to have two programs that are algorithmically equivalent but not formally equivalent and vice versa. What we have then is not a strict hierarchy in equivalence relations but a partial ordering. Denoting formal identity by I, formal equivalence by E, algorithmic equivalence by A, and functional equivalence by F, this can be expressed symbolically as follows:

$$I \subset E \subset F$$
$$I \subset A \subset F$$
$$\text{not } E \subset A$$
$$\text{not } A \subset E$$

For the third of the four equivalence relations, two programs are considered equivalent if they use the same algorithm regardless of formal characteristics of the programs themselves. Thus, algorithmic equivalence is concerned with the dynamics of the programs, whereas formal identity and formal equivalence were concerned with static qualities. For our purposes the algorithm used in a program is determined by the values taken on by real variables, the output, and the sequence in which these occur. The names used for the variables are immaterial and we will be concerned only with those variables that take on real numbers as values. Hence, we will be interested in the values of indexed variables, such as X(3) and A(I,J), but not in user-defined functions or in forms. As for output, we will generally be concerned only with computed numeric values and not with the content of whatever text is also output; thus, we will consider any two text strings to be equivalent except for the few problems (e.g., L15-21) for which the only expected output is text, and in those cases we will consider two text strings to be identical if their content has the same (English) meaning.

To clarify this notion we will represent the stored data at any point in time as an n-tuple of the values of the n variables to which values have been assigned. The order of the numbers in an n-tuple is dependent upon the order in which the variables were first given values; thus, the first number in the n-tuple is the current value of the first variable to which any value was assigned, etc. As an example, consider the following simple program:

<u>Example 1.</u>

```
1.1  SET I = 1
1.2  SET X = I*2
1.3  TYPE X
1.4  SET I = I+1
1.5  TO STEP 1.2 IF I < 3
```

The first variable to be used by this program is I, so the value of I will always appear as the first number in the n-tuple representing the stored data. These n-tuples, in the order in which they occur, are

(1)

(1,2)

(2,2)

(2,4)

(3,4)

We will also be interested in the output, and how it fits into the above sequence, and will represent the sequence of stored data and output as follows:

(1)

(1,2)

Output: 2

(2,2)

(2,4)

Output: 4

(3,4)

The above sequence represents what we will call the <u>algorithm</u> for the program in Example 1. Example 2 is another program which differs from Example 1 only in the IF clause used in the fifth step.

<u>Example 2.</u>

```
2.1  SET I = 1
2.2  SET X = I*2
2.3  TYPE X
2.4  SET I = I+1
2.5  TO STEP 2.2 IF I <= 2
```

If we write the algorithm for Example 2, we find it to be identical with that for Example 1. Hence, the two programs are algorithmically equivalent. Notice, however, that these two programs are not formally equivalent since the expressions $I < 3$ and $I \leq 2$ are not logically equivalent. Our third example is a program that is formally equivalent to Example 1 but not algorithmically equivalent. (As we will see later, all three programs are functionally equivalent.)

Example 3.

```
3.1 SET I = 1
3.2 SET X = I*2
3.3 SET I = I+1
3.4 TYPE X
3.5 TO STEP 3.2 IF I < 3
```

This program is simply Example 1 with the third and fourth steps interchanged. By Rule 12 for formal equivalence, we find Examples 1 and 3 to be formally equivalent. However, the algorithm for Example 3 is

```
(1)
(1,2)
(2,2)
Output: 2
(2,4)
(3,4)
Output: 4
```

which is not identical to the algorithm for Example 1.

The examples above are too simple to fully illustrate the concept of algorithmic equivalence since they do not use input data. Following is a simple example of a program that uses a single numeric input.

Example 4.

```
4.1 SET Y = X IF X >= 0
4.2 SET Y = -X IF X < 0
4.3 TYPE Y
```

For this program the sequence of stored data and output depends upon the value preassigned to X, the input variable. For example, if X is

-5 the sequence is

(-5)

(-5, 5)

Output: 5

For each value of X there is a different sequence and it is the entire set of such sequences that determines the algorithm.

Algorithmic equivalence, as defined above, is slightly more powerful for the set of programs under consideration than formal equivalence. There are an average of 3.6 equivalence classes per problem, as compared to the five equivalence classes for formal equivalence and the eight classes for formal identity. There is an average of 6.1 programs per class, whereas formal equivalence yields 4.3 programs per class and formal identity 2.7 programs per class. The classes under algorithmic equivalence are labeled A_1, A_2, etc., in Appendix A, which lists the classification of every program in the data.

The fourth, and last, equivalence relation used in classifying student-written programs is <u>functional equivalence</u>. In determining the function of a program we consider only the output and not the form of the program or the values of any variables other than input and output variables. As with algorithmic equivalence, text in which numeric results are imbedded is ignored; only for those few programs whose output is non-numeric do we consider the text that is printed. Also, as for algorithmic equivalence, numeric results are rounded to three significant digits. Functional equivalence is the simplest and the most powerful of the four methods of classification. There are an average of 1.7 equivalence classes per problem, with an average of 13.1 programs in each class. For 14 of the 25 problems all of the student-written programs were functionally equivalent.

From the wide variety of possible methods of classification for programs, we have chosen four to study in some depth. In choosing these four methods, we have been guided by the following considerations. (1) We wanted to use equivalence relations that conformed to intuitive notions of program equivalence. (2) The equivalence relations should show considerable spread in their "grouping power" over the set of data; that is, the weakest of the relations should provide only a minor reduction in the number of types, whereas the strongest should come close to grouping all programs into a single class. (3) The equivalence relations should be mathematically defensible, in that the concept of equivalence is well-defined independent of the data. In relation to (3) we would also have preferred to exhibit equivalence relations for which a decision procedure could be defined. Except for I, formal identity, our definitions do not satisfy this requirement, and we saw no way of satisfying this without seriously violating either (1) or (2). By defining formal equivalence more strictly, in particular by suitably restricting substitution of algebraic expressions, we could have provided a definition that would admit of a decision procedure. For future studies we recommend that this approach be explored in more depth. Our recommendation for this is based on the feeling that formal equivalence most nearly approaches the intuitive notion of equivalence expressed by students in phrases such as "these two programs are really the same" or "these two programs may do the same thing but they do it quite differently." A programming consultant, automated or human, who is trying to help a student complete a partially written program or debug a faulty program, would be ~~most~~ likely to be effective if he (or it) can guide the student towards a formally equivalent correct solution. An automated consultant could do this only if it were capable of determining to which formal equivalence class the student's partial solution belonged. In other words, the consulting routine would need a decision procedure for a formal equivalence relation that extended to incomplete and incorrect solutions as well as correct solutions.

Although three of our four equivalence relations do not admit of decision procedures, requirements (2) and (3) were well satisfied. Whether or not requirement (1)--that the equivalence relations are intuitively valid--is satisfied is left to the reader to decide. In connection with this we mention several other possible means of classification that could have been used. Formal identity and formal equivalence, for instance, are but two of a very large number of equivalence relations based on substitution of semantically equivalent parts of programs. Any one of the substitution rules defined above, or any arbitrary set of those rules, would define an equivalence relation. There are also a large number of applicable substitution rules that we did not list. One, for example, would allow the permutation of two adjacent DEMAND commands. Another would allow the substitution of

```
        SET x = e
        TYPE x
```

for

```
        TYPE e
```

where x is a variable that does not occur in e or elsewhere in the program. One could also devise more complicated rules of substitution, such as the substitution of iterated subroutines for certain kinds of loops. We did not use the last-mentioned of these possibilities because we felt that such a substitution would allow us to equate programs that students feel to be quite different. As for the other two possibilities above, we did not use them (and many similar rules) because there was no instance in the data where they could be applied; all of the 15 substitution rules listed for formal equivalence were actually used in classifying the data.

Besides the many kinds of formal equivalence relations that could be used, there are a number of possible variants on algorithmic and functional equivalence. We might, for example, have differentiated programs on the basis of the output text; in studying programs written in languages with string manipulation features, such distinctions would be of more importance. In defining algorithmic equivalence, we considered the sequence of values for all variables used by the program; for more complex programs than those found in the data analyzed here, it might be wise to exclude variables bound in subroutines or even variables bound in simple loops. For block structured languages only global variables might be considered. As for functional equivalence, a more powerful equivalence relation could be defined by considering functions to be equivalent if they differed by at most a fixed number of values.

In summary, out of the wide variety of possible, well-defined equivalence relations, we chose four that were sufficiently different to illustrate the spectrum of possibilities, guided in our choice to a large extent by intuitive appeal. In the next chapter we will analyze the effects of each of these equivalence relations on the given set of data.

CHAPTER VIII

Diversity of Solutions

In looking at programs written by students (Appendix B) one is struck by the fact that for some problems most students produced very similar looking programs, whereas for others there seem to be few points of similarity. In this chapter we devote ourselves to the study of the diversity of programs written by students and attempt to explain why there is more diversity for some problems than for others. To do this we will first introduce a suitable measure of diversity and then investigate the statistical relationship between diversity and various measurable qualities of the problems and the curriculum.

The amount of diversity observed in a set of solutions to a given problem is dependent not only upon the solutions themselves but upon one's notion of similarity, or equivalence. Thus, for different equivalence relations the observed diversity may be different even for the same set of data. In this study we are concerned only with the four concepts of program equivalence discussed in the preceding chapter, and as a result will have four different definitions for diversity: diversity of function, diversity of algorithm, diversity of equivalent forms, and diversity of identical forms (where, by "identical" we mean formally identical, as defined in Chapter VII).

Since the measure of diversity used here is not widely known, we discuss it briefly before describing the statistical analyses. A more complete and precise mathematical discussion of the measurement of diversity is given in Appendix C.

Suppose $\mathcal{S}$ is a population that is partitioned into k classes, and that the probability that an element is in the i-th class is p_i for each $i = 1,2,\ldots,k$. The diversity of $\mathcal{S}$, for the given method of classification, is

$$\delta = 1 - \sum_{i=1}^{k} p_i^2 .$$

The value of δ will be between 0 and 1, and will be 0 only if all elements of $\mathcal{S}$ are in a single class. For a fixed value of k, the largest value of δ occurs when the p_i's are equal, that is, when the members of $\mathcal{S}$ are evenly distributed among the classes. For an even distribution, the value of δ increases with an increasing number of classes, approaching 1 as a limit.

Let S be a sample* of size N from the population $\mathcal{S}$, and let n_i be the observed number of occurrences of the i-th class for $i = 1,2,\ldots,k$. Then p_i can be estimated by $\frac{n_i}{N}$, and it would be natural to define the sample statistic for diversity to be

$$d = 1 - \sum_{i=1}^{k} \left(\frac{n_i}{N}\right)^2 .$$

It transpires, however, that as an estimator of δ, the statistic d is biased, so we define

$$\hat{d} = \frac{N}{N-1} d$$

*We consider only unordered samples with replacement.

as the estimator of the parameter δ. $\hat{d}$, as shown in Appendix C, is a consistent, unbiased estimator of δ.

A little algebraic manipulation will show that

$$\hat{d} = 1 - \sum_{i=1}^{k} \frac{n_i(n_i-1)}{N(N-1)} \ .$$

From this formulation we see that $\hat{d}$ attains a maximum of 1 whenever each member of S is in a separate class, that is, whenever each n_i is 1. We also see that the minimum of $\hat{d}$ is 0, which occurs whenever all members of the set are in the same class.

It is evident that diversity is independent of the indices used for the p_j's or n_i's. For example, any reordering of the subscripts of $n_1, n_2, \ldots, n_k$ would not change the calculated value of d. Thus, diversity is invariant under any 1-1 transformation of the indices, which is all that is required to assure that the formula is appropriate for categorical scales of measurement.

Although we have shown that $\hat{d}$ is appropriate as a statistic, it remains to be shown that this formula is an appropriate measure of diversity. In regards to this question the germane property of $\hat{d}$ is that it is the probability that two elements drawn at random will not be equivalent. As a result, if we remove one element from a more numerous class and place it in a less numerous (perhaps empty) class, thereby increasing the diversity, the value of $\hat{d}$ is increased.

As mentioned, the value of diversity is dependent upon the underlying equivalence relation. For the same set of data an equivalence relation with more grouping power will produce a lower value for $\hat{d}$ than a less powerful equivalence relation. To emphasize the dependence of diversity on the equivalence relation we will denote diversity of function by $\hat{d}_F$, diversity of algorithm by $\hat{d}_A$, etc.

Let us now turn to the data to find the amount of diversity for each problem using four measures of diversity, one for each of the four equivalence relations I, E, A, and F. In Table 20, the statistics for I, formal identity, are shown; the number of solutions in each equivalence class is listed for each problem, and the value of $\hat{d}_I$ is shown in the last column. The corresponding statistics for the equivalence relations E, A, and F are shown in Tables 21, 22, and 23. For formal identity, the diversity ranges from 0.12 to 1.00 with a mean of 0.76. For formal equivalence the range is even larger--from 0 to 1.00--and the mean is 0.56. Thus, the diversity of identical forms is nearly 50% greater than the diversity of equivalent forms. Even with this sizable change in the average diversity, there are five problems for which there is no difference between $\hat{d}_I$ and $\hat{d}_E$ and another five for which the difference is less than 0.1; for these 10 problems essentially all of the diversity in form is due to the trivial variations allowed under formal identity. Comparing the values of $\hat{d}_I$ and $\hat{d}_E$ problem by problem, we note that $\hat{d}_E$ is never greater than $\hat{d}_I$, a necessary consequence of the fact that formal identity is included in formal equivalence.

The average value of $\hat{d}_A$ (diversity of algorithm) is 0.43, only slightly smaller than the 0.56 average for $\hat{d}_E$. For eight of the 25 problems the values of $\hat{d}_E$ and $\hat{d}_A$ are identical, indicating a strong relationship between the grouping powers of algorithmic equivalence and formal equivalence. This is in marked contrast to the relationship of

Table 20

Number of Solutions in Each Equivalence Class, Using the Equivalence Relation of Formal Equivalence

Problem Number	Equivalence Class														Number of Programs	Number of Equivalence Classes	Diversity* $\hat{d}_I$
	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}			
L5-30	19	6	2	2	2	1	1								33	7	0.64
L8-9	18	15	2	1											36	4	0.59
L8-27	7	5	4	3	2	2	1	1	1	1	1	1			29	12	0.90
L8-28	8	3	3	2	2	2	1	1	1	1	1	1			26	12	0.89
L9-3	6	5	3	3	2	1	1	1	1						23	9	0.87
L9-8	11	8	3	2	1	1									26	6	0.73
L10-12	15	11	1	1	1	1	1	1	1	1					27	10	0.68
L10-19	18	11	1												30	3	0.52
L11-11	6	5	4	2	1	1	1	1	1	1	1	1	1		26	13	0.90
L12-4	32	1	1												34	3	0.12
L13-29	11	5	3	2	1	1	1	1							25	8	0.77
L15-15	4	3	2	1	1	1	1	1	1	1	1	1	1	1	20	14	0.95
L15-17	13	5	1	1	1	1	1								23	7	0.65
L15-18	6	3	2	1	1	1	1	1	1	1					18	10	0.88
L15-21	17	4	1	1	1	1									25	6	0.53
L16-4	11	7	2	1	1	1	1	1	1	1	1	1			29	12	0.81
L16-6	1	1	1	1	1	1	1	1							8	8	1.00
L23-7	2	2	1	1	1	1	1	1	1	1	1				13	11	0.97
L24-11	11	2	2	1	1	1									18	6	0.63
L25-8	15	4	2	2											23	4	0.55
L26-5	6	5	2	1	1	1	1	1	1	1	1	1			22	12	0.89
L29-19	2	1	1	1	1										6	5	0.93
L32-5	2	1	1	1	1	1	1	1							9	8	0.97
L32-8	7	2	1	1	1	1	1	1	1						16	9	0.82
L32-19	2	1	1	1	1										6	5	0.93
																Mean:	0.76
																S.D.:	0.20

Table 20 (cont'd)

*The diversity of solutions for each problem is calculated by the formula

$$\hat{d}_I = \frac{N^2 - \sum_{i=1}^{c} n_i^2}{N(N-1)}$$

where n_i is the number of programs in Equivalence Class I_i, N is the total number of programs, and c is the number of equivalence classes.

Table 21

Number of Programs in Each Equivalence Class, Using the Equivalence Relation of Formal Equivalence

Problem Number	Equivalence Class											Number of Programs	Number of Equivalence Classes	Diversity*
	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}			$\hat{d}_E$
L5-30	22	6	2	2	1							33	5	0.53
L8-9	36											36	1	0
L8-27	15	5	5	2	2							29	5	0.69
L8-28	22	4										26	2	0.27
L9-3	20	2	1									23	3	0.25
L9-8	21	3	1	1								26	4	0.34
L10-12	26	1										27	2	0.07
L10-19	30											30	1	0
L11-11	10	7	7	1	1							26	5	0.73
L12-4	32	1	1									34	3	0.12
L13-29	13	5	3	1	1	1	1					25	7	0.70
L15-15	4	3	2	2	2	2	1	1	1	1	1	20	11	0.93
L15-17	13	6	1	1	1	1						23	6	0.63
L15-18	7	6	2	1	1	1						18	6	0.76
L15-21	18	6	1									25	3	0.44
L16-4	13	9	2	2	1	1	1					29	7	0.71
L16-6	1	1	1	1	1	1	1	1				8	8	1.00
L23-7	2	2	1	1	1	1	1	1	1	1	1	13	11	0.97
L24-11	11	2	2	1	1	1						18	6	0.63
L25-8	23											23	1	0
L26-5	11	3	2	2	1	1	1	1				22	8	0.74
L29-19	2	1	1	1	1							6	5	0.93
L32-5	2	2	1	1	1	1	1					9	7	0.94
L32-8	8	2	1	1	1	1	1	1				16	8	0.76
L32-19	3	1	1	1								6	4	0.80
													Mean:	0.56
													S.D.:	0.33

Table 21 (cont'd)

*The diversity of solutions for each problem is calculated by the formula

$$\hat{d}_E = \frac{N^2 - \sum_{i=1}^{c} n_i^2}{N(N-1)}$$

where n_i is the number of programs in Equivalence Class E_i, c is the number of equivalence classes, and N is the total number of programs.

Table 22

Number of Solutions in Each Equivalence Class, Using the Equivalence Relation of Functional Equivalence

Problem Number	Equivalence Class								Number of Programs	Number of Equivalence Classes	Diversity*
	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8			$\hat{d}_A$
L5-30	22	6	2	2	1				33	5	0.53
L8-9	36								36	1	0
L8-27	20	4	3	2					29	4	0.51
L8-28	22	4							26	2	0.27
L9-3	20	2	1						23	3	0.25
L9-8	21	3	1	1					26	4	0.34
L10-12	27								27	1	0
L10-19	30								30	1	0
L11-11	10	8	4	2	1	1			26	6	0.75
L12-4	33	1							34	2	0.06
L13-29	19	3	1	1	1				25	5	0.42
L15-15	12	8							20	2	0.51
L15-17	15	7	1						23	3	0.50
L15-18	15	2	1						18	3	0.31
L15-21	25								25	1	0
L16-4	23	4	1	1					29	4	0.36
L16-6	4	1	1	1	1				8	5	0.61
L23-7	5	3	1	1	1	1	1		13	7	0.83
L24-11	14	2	2						18	3	0.39
L25-8	23								23	1	0
L26-5	11	5	3	1	1	1			22	6	0.71
L29-19	3	1	1	1					6	4	0.80
L32-5	2	1	1	1	1	1	1	1	9	8	0.97
L32-8	8	2	2	2	1	1			16	6	0.74
L32-19	3	1	1	1					6	4	0.80
										Mean:	0.43
										S.D.:	0.31

Table 22 (cont'd)

*The diversity of solutions for each problem is calculated by the formula

$$\hat{d}_A = \frac{N^2 - \sum_{i=1}^{c} n_i^2}{N(N-1)}$$

where n_i is the number of programs in Equivalence Class A_i, c is the number of equivalence classes, and N is the number of programs.

Table 23

Number of Programs in Each Equivalence Class, Using the Equivalence Relation of Functional Equivalence

Problem Number	Equivalence Class					Number of Programs	Number of Equivalence Classes	Diversity*
	F_1	F_2	F_3	F_4	F_5			$\hat{d}_F$
L5-30	33					33	1	0
L8-9	36					36	1	0
L8-27	29					29	1	0
L8-28	26					26	1	0
L9-3	23					23	1	0
L9-8	25	1				26	2	0.08
L10-12	27					27	1	0
L10-19	30					30	1	0
L11-11	26					26	1	0
L12-4	34					34	1	0
L13-29	23	2				25	2	0.15
L15-15	12	8				20	2	0.51
L15-17	23					23	1	0
L15-18	17	1				18	2	0.11
L15-21	25					25	1	0
L16-4	28	1				29	2	0.07
L16-6	7	1				8	2	0.25
L23-7	11	1	1			13	3	0.29
L24-11	18					18	1	0
L25-8	23					23	1	0
L26-5	20	2				22	2	0.17
L29-19	6					6	1	0
L32-5	5	1	1	1	1	9	5	0.72
L32-8	8	7	1			16	3	0.59
L32-19	3	2	1			6	3	0.73
							Mean:	0.15
							S.D.:	0.24

Table 23 (cont'd)

*The diversity of solutions fcr each problem is calculated by the formula

$$\hat{d}_F = \sqrt{\frac{c}{\sum_{i=1}^{c} n_i^2}} \times 100$$

where n_i is the number of programs in Equivalence Class F_i, and c is the number of equivalence classes.

either of these relations to functional equivalence. For functional equivalence the average diversity is 0.15, about one-third the value of the average of $\hat{d}_A$, even though six of the values of $\hat{d}_A$ and $\hat{d}_F$ are equal. For a more precise comparison of the grouping powers of the four equivalence relations, see the correlation matrix for $\hat{d}$ given in Table 24. All correlations are high, as might be conjectured from the logical relationships between the equivalence relations. The values of r are all greater than 0.47 and the highest value is 0.88 for the correlation between algorithmic equivalence and formal equivalence.

In scanning the various values of $\hat{d}$, we note an increase in diversity with problem number. This is most noticeable in the case of functional equivalence but is also true for the other three relations. From this we conjecture that students' programs tend to become more diverse as the students gain experience. The relationship is far from perfect, however, indicating that other variables also have effect on the amount of diversity. It seems reasonable to suppose that certain problems lend themselves more readily to a variety of solutions. Look, for example, at the values of $\hat{d}$ for Problem L32-19; $\hat{d}_I = 0.93$, $\hat{d}_E = 0.80$, $\hat{d}_A = 0.80$, and $\hat{d}_F = 0.73$, all of which are well above the averages for the respective equivalence relations. Problems L16-6 and L23-7 also have very high values of $\hat{d}$ for all four equivalence relations. The question is, do these problems have similar characteristics that might account for such a wide variety of solutions. On the other hand, it may be that the method of instruction, rather than the problem itself, influences the amount of diversity. Some of the 25 problems contained quite strong suggestions about the form a

Table 24

Correlation Matrix for Four Equivalence Relations

	I	E	A	F
I	1.000	0.721	0.728	0.476
E		1.000	0.884	0.561
A			1.000	0.624
F				1.000

I = Formal Identity

E = Formal Equivalence

A = Algorithmic Equivalence

F = Functional Equivalence

correct solution might take; if students follow these suggestions closely we would expect their solutions to be very similar.

To investigate these and other conjectures we ran four step-wise multiple linear regressions, using $\hat{d}_I$, $\hat{d}_E$, $\hat{d}_A$, and $\hat{d}_F$ as the variables to be predicted. For independent variables we used the 10 independent variables described in Chapter VI: IF, ARG, FCT, REIT, LNG, INPT, LES, HELP, VOC, and NEW. Recall that these 10 variables measure characteristics of the problems, the curriculum, and the expected correct solutions (listed in Chapter II), and are independent of the data.

We have already discussed, in Chapter VI, the correlation coefficients for the various pairs of independent variables. Before giving the results of the linear regressions let us look at the correlations between the independent variables and the amount of diversity. The correlation coefficients are shown in Table 25. The first point of interest is that there are a large number of quite high values; 11 of the coefficients have (absolute) values greater than .5. Secondly, the signs of the coefficients are constant across the different definitions of diversity; from our knowledge of the logical and statistical relationships between the four equivalence relations this is not surprising. We next note that the three independent variables that correlate most highly with diversity (on the average) are LES, REIT, and VOC. As we noted before, these three variables are also highly correlated with one another ($|r| > .6$). Although these three variables correlate with diversity most highly on the average, it is not the case that they correlate most highly with any given measure of diversity. Both IF and HELP are more highly correlated with $\hat{d}_I$ than any of LES, REIT, or VOC. IF and LNG are the most highly correlated variables for $\hat{d}_E$.

Table 25

Correlations Between Independent Variables and Four Measures of Diversity

Independent Variable	Measure of Diversity $\hat{d}_I$	$\hat{d}_E$	$\hat{d}_A$	$\hat{d}_F$
IF	0.341	0.559	0.286	0.189
ARG	0.148	0.286	0.307	0.583
FCT	0.262	0.242	0.412	0.114
REIT	0.292	0.428	0.568	0.658
LNG	0.189	0.575	0.416	0.065
INPT	-0.067	-0.423	-0.239	-0.204
LES	0.306	0.501	0.566	0.664
HELP	-0.335	-0.355	-0.389	-0.022
VOC	0.235	0.544	0.501	0.570
NEW	-0.250	-0.262	-0.264	-0.085

It is also of some interest to look at the pairs for which the correlation is low. The average values of r for FCT, INPT, NEW, and HELP are all less than 0.3. Also the correlation between LNG and $\hat{d}_F$ is less than 0.1.

A more revealing picture of the relationships between the independent variables and the four measures of diversity is given by the results of the multiple regressions. The derived linear models are given in Table 26, and a summary of the step-wise regressions is given in Table 27. The amount of variance in diversity accounted for varies from 56% to 80%, the best fit being for functional equivalence and the poorest for formal identity. Thus, using the same set of independent variables, we obtain somewhat better predictions of diversity than of problem difficulty (compare Table 18). If we look only at the amount of variance accounted for by the first five variables to enter the regressions, we find that we can account for over 60% of the variance for three of the four measures of diversity (excluding $\hat{d}_I$).

The order in which the independent variables entered into the regressions is different in all four cases. But it is instructive to note that for $\hat{d}_I$, $\hat{d}_E$, and $\hat{d}_A$ the first five variables to enter are identical. For these three cases the first five variables are IF, FCT, REIT, LNG, and HELP. Three of these, IF, REIT, and LNG, are among the first five variables to enter into the regression for $\hat{d}_F$. Taking all things into consideration it is probable that REIT and IF are the two most effective variables for the prediction of diversity. For both of these the relationship is direct, that is, an increase in the value of REIT or of IF will cause an increase in diversity. Thus, in predicting diversity, the

Table 26

Linear Models for the Prediction of Diversity

$$\hat{d}_I \times 100 = 68 + 97\,\text{IF} - 7\,\text{ARG} + 20\,\text{FCT} + 8\,\text{REIT} - 2\,\text{LNG} - 0.2\,\text{INPT} + 3\,\text{LES} - 3\,\text{HELP} - 8\,\text{VOC} - 4\,\text{NEW}$$

$$\hat{d}_E \times 100 = 24 + 87\,\text{IF} + 3\,\text{ARG} + 24\,\text{FCT} + 15\,\text{REIT} - 1\,\text{LNG} - 3\,\text{INPT} - 0.4\,\text{LES} - 10\,\text{HELP} + 1\,\text{VOC} - 10\,\text{NEW}$$

$$\hat{d}_A \times 100 = 4 + 60\,\text{IF} + 26\,\text{FCT} + 26\,\text{REIT} + 0.9\,\text{LNG} - 0.4\,\text{INPT} + 3\,\text{LES} - 8\,\text{HELP} - 6\,\text{VOC} - 6\,\text{NEW}$$

$$\hat{d}_F \times 100 = -27 + 62\,\text{IF} + 11\,\text{ARG} + 5\,\text{FCT} + 30\,\text{REIT} - 3\,\text{LNG} + 1\,\text{INPT} + 0.8\,\text{LES} + 10\,\text{NEW}$$

Table 27

Amount of Variance in Diversity Accounted for by Linear Combinations of Ten Independent Variables

Step Number	Dependent Variable											
	$\hat{d}_I$			$\hat{d}_E$			$\hat{d}_A$			$\hat{d}_F$		
	Variable Entered	Variable Removed	R^2	Variable Entered	Variable Removed	R^2	Variable Entered	Variable Removed	R^2	Variable Entered	Variable Removed	R^2
1	IF		.116	LNG		.330	REIT		.322	LES		.441
2	FCT		.243	REIT		.474	LNG		.458	ARG		.564
3	REIT		.308	IF		.559	HELP		.543	REIT		.663
4	LNG		.391	FCT		.631	FCT		.600	LNG		.685
5	HELP		.438	HELP		.666	IF		.641	IF		.748
6	NEW		.462	INPT		.701	ARG		.655	NEW		.774
7	VOC		.470		LNG		NEW		.668	INPT		.791
8	LES		.536	NEW		.712	VOC		.670	FCT		.797
9	ARG		.567	ARG		.715	LES		.696	VOC		.800
10	INPT		.568	VOC		.715		ARG				
11				LNG		.716	INPT		.697			
12				LES		.717						
Total variance accounted for	56.8%			71.7%			69.7%			80.0%		

most influential variables are (1) whether or not loops or subroutines are required, and (2) what proportion of the commands are likely to be conditional. Because of the importance of conditionals, loops, and subroutines in programming, this is not a surprising result. The third most influential variables in predicting diversity is LNG, the length of the expected correct solution. All three of these variables (REIT, IF, and LNG) are based on characteristics of the expected correct solutions. To some extent they depend upon the context of the problem, but they are more largely dependent upon the problem per se. In contrast, two of the four most influential variables in the prediction of problem difficulty are LES and HELP, both of which are entirely dependent upon the curriculum.

Although LES made little contribution to the first three diversity regressions, it did enter first into the regression for $\hat{d}_F$, accounting for 44% of the variance in diversity of function. Thus, at least one of the diversity predictions is highly dependent upon a curriculum variable. HELP, however, did not enter into the $\hat{d}_F$ regression at all. In the other three diversity regressions HELP was among the first five variables but did not contribute as much, on the average, as the other four variables (IF, FCT, REIT, LNG).

In summary, IF and REIT are the two most important variables for the prediction of diversity and problem difficulty. Further, the prediction of diversity depends more upon the programming problem itself than upon the context in which the problem is found, unlike problem difficulty, which is more dependent upon curriculum context; the exception is in the prediction of diversity of function to which LES made a substantial contribution.

We began this discussion with the conjecture that diversity increases with problem number. The correlations between LES and the four measures of diversity (average $r > .5$), tend to substantiate this view. However, the more penetrating analysis provided by the multiple regressions throws this conjecture into doubt for three of the four measures of diversity ($\hat{d}_I$, $\hat{d}_E$, and $\hat{d}_A$) for which we can conclude that the implied relationship with LES is spurious and is a result of the merely statistical relationship between LES and more effective variables such as REIT and LNG. For the fourth measure of diversity the multiple regression confirmed the conjecture that LES is closely related to diversity. We feel that a deeper analysis might provide evidence for the rejection of this hypothesis if another curriculum or other sets of programming problems were chosen for study.

We also conjectured that the HELP variable would be important in predicting diversity since it measures whether or not a nearly equivalent program is displayed as an example on which students may model their solution. Unexpectedly, the linear regressions did not support this conjecture strongly. For only one measure of diversity did HELP play an effective part. For algorithmic equivalence HELP entered into the regression at the third step, increasing the value of r^2 by 8%. For functional equivalence, HELP did not enter into the regression at all, indicating a contribution of less than 0.1%. Since HELP is a curriculum variable that is clearly under the control of the curriculum designer, it might be fruitful to conduct a future experiment in which different treatments are used for different groups of students.

Before closing the discussion of diversity we want to present one further comparison between diversity and problem difficulty. For problem difficulty we use only our primary measure, M1, the proportion of correct responses given on the first trial. Following are the correlations between M1 and each of the four measures of diversity:

$\hat{d}_I$: -0.584

$\hat{d}_E$: -0.710

$\hat{d}_A$: -0.725

$\hat{d}_F$: -0.463

These correlation coefficients are all negative, indicating that diversity increases with difficulty (recall that proportion correct is inversely related to problem difficulty, by definition). Furthermore, all of the values are substantial; for $\hat{d}_E$ and $\hat{d}_A$, in particular, we could account for half of the variance in diversity from a knowledge of difficulty.

In one sense these correlations are misleading. It would be easy to fall into the trap of assuming that students would do better if less diversity were allowed by the curriculum. Assuming that we could control the amount of diversity, it seems likely from the results of the multiple regressions that we could do this only by changing the problems themselves rather than the way in which they were presented; this conclusion is based on the fact that diversity seems to depend more upon problem variables (IF, REIT) than upon curriculum variables (LES, HELP). Thus, we could decrease the diversity, and increase the proportion correct, by giving fewer problems that required the use of conditions or loops. If we did this, would we thereby increase the total learning of programming?

We do not intend to pursue this further here and have only mentioned the relationship between difficulty and diversity to point out that the question of trade-offs is a complex and subtle one that needs study in much greater depth.

CHAPTER IX

Summary

In this paper we have presented a detailed study of 747 computer programs written by 40 students in response to 25 programming problems given in a computer-assisted course in programming. The 25 problems vary widely in kind and difficulty. Some of them are no more than "finger exercises" designed solely to provide the student with on-line practice in the use of newly introduced syntactic features of the programming language. Other problems are sufficiently complex logically that an experienced programmer must use care to arrive at a correct solution. The standard solutions to the 25 problems (which we called "expected correct solutions") varied in length from 2 to 13 commands. Ten of the 25 problems required the use of one or more conditional commands, and six of them required the use of either subroutines or loops. Most of the problems required a program that performed only one mathematical function but three of them required more than one function to be performed on the input data. For most problems test values (for input) were specified in the problem statements; however, for eight problems either no input was required or the students were free to choose appropriate values to use in testing their programs.

Eleven of the problems required the use of a newly introduced lexical item or syntactic feature of the language. The curriculum offered varying amounts of guidance to the student. For four of the 25 problems, a complete similar program was shown to the student as a model from which he could work; an additional seven problems displayed a part of a program that could serve as a model.

The above mentioned characteristics of the problems and curriculum context were used as independent variables for step-wise multiple regressions (to be discussed below). Other independent variables used in the regressions measured the position of the problem in the curriculum, the amount of AID vocabulary so far introduced in the course, and the number of arguments required by the mathematical function performed by the program.

Since students were not required to try to solve every problem, we did not expect to find 40 (students) × 25 (problems) = 1000 attempted solutions. We did find that a high proportion of the students attempted to solve the problems--75% on the average. Some of the attempts made by students were cursory, but on the whole we concluded that most made a serious effort; the total number of commands given by students was 7063* and the average number of commands given by the students who attempted the problem was nearly twice the number needed for an economical solution to the problem.

Many of the commands given by students were in error, and, in fact, a large number (26%) were never executed, either because they contained errors that caused an execution error or because the student made no effort to use the commands. (He may, for example, have replaced the command with another before he executed his program.)

In studying the kinds of commands given by students, we classified the commands according to the AID verb used and found that we could predict the proportions of the types of commands quite well simply from the

*on first trial

corresponding proportions found in our expected correct solutions. The amount of variance accounted for by the simple linear model was over 90%.

When the commands given by students were classified as either direct or indirect, we found that nearly half (45%) of the students' commands were direct; in comparison, only 28% of the commands in the expected correct solutions were direct.

In a study of the distribution of correct solutions, we found that 57% of the attempts were successful on first trial. From other studies we know that the average for all exercises in the course is greater than 75%, and thereby conclude that the set of problems chosen for study here were considerably more difficult than the average exercise in the course.

In addition to a simple correct-incorrect system of grading, we used a method of assigning partial credit based on the number of commands used in a correct or partially correct solution. We found that, on the average, fewer than half of the typed commands contributed to a correct solution. Of the commands that were executed, two-thirds contributed toward a correct solution.

For a third measure of correctness, we used a correct-incorrect classification in which errors in algebraic formulas were disregarded. The average proportion correct "up to" algebraic errors was 62%, as compared to the 57% for a strict correct-incorrect measure. Although the correlation between these two measures of correctness was quite high ($r = .90$), there were several problems for which the differences were extreme.

A detailed analysis of errors was also undertaken. We found 1090 overt errors in the 7063 commands given by students. Two-thirds of these were syntax errors and one-third were semantic. In looking at syntax errors we found the most numerous errors (about one-third) to be either typographical errors or incomplete commands. We also found that a disturbingly high proportion of the syntax errors (perhaps 20%) were what we called "errors of overgeneralization," that is, errors that were apparently caused by an overgeneralization of the syntax rules. These errors were reasonable constructions in the sense that the intended meaning was perfectly clear; in fact, in most cases, these erroneous commands could have been parsed by a slightly more sophisticated interpreter. It is known that children, in learning natural languages, overgeneralize on the rules that govern the syntax and usage of that language. Such examples range from generalizing the rules for creation of inflections ("goed" as the past of "to go" instead of the irregular but correct "went"), to structural errors and the misapplication of constructions in pragmatic contexts. The high frequency of the same type of error in this study, concerned with the learning of a formal as opposed to a natural language, suggests that there are common governing principles for the acquisition of both. An awareness of the tendency of students to overgeneralize from specific rules of syntax could enable programmers to produce high level languages that could be more readily learned. For example, if the AID interpreter allowed the use of multiple arguments with SET and DEMAND exactly as it did for TYPE and DELETE, many errors would have been avoided.

In studying semantic errors we found a rather large proportion (over 20%) of algebraic errors. Some of these errors seemed to stem from ignorance of the correct algebraic formulas and some from incorrect translation into AID notation. Most incorrect translations were the result of a poor understanding of the hierarchy of operations. Dummy variables and their use in the definitions of functions also gave rise to a number of errors. In general, logical errors, either in IF clauses or in the sequence of execution, were fewer than anticipated. From this evidence we concluded that in all likelihood students are less mathematically sophisticated than presumed by the curriculum; consequently, in a subsequent revision of the course we included more instruction in mathematics and delayed the introduction of user-defined functions until quite late in the course. A future comparison of semantic errors for the two versions of the course would be needed to establish the accuracy of our conclusion that much of the difficulty is curriculum-oriented and can be controlled by the curriculum writer.

The descriptive statistics mentioned above were used in defining 19 different measures of problem difficulty. Three were measures of proportion correct. Ten were measures of number of errors and error rates, for both syntax and semantic errors as well as total errors. Five measures of problem difficulty were measures of the effort expended and the final measure was the proportion of students who attempted the problem. In comparing these 19 measures we found that some pairs, such as the first two measures of proportion correct, were highly correlated, but there were many pairs for which the correlation coefficient was essentially zero, leading us to conclude that the measurements are along several different dimensions of problem difficulty.

We selected six of the 19 measures of problem difficulty for more intense study. These were (1) the proportion correct on first trial,

(2) the proportion correct up to algebraic errors, (3) the syntax error rate, (4) the number of students who made semantic errors, (5) the ratio of commands typed to the number of commands needed for a correct solution, and (6) the number of students who attempted the problem. Except for the first two of these six measures, the correlations between pairs were quite low. By means of step-wise multiple linear regressions we derived linear models that predicted problem difficulty from measurable characteristics of the problems, the standard solutions, and the curriculum context. The same set of 10 independent variables was used in each of the six regressions. These 10 variables measured such characteristics as the expected proportion of conditional commands (IF), the location of the problem in the curriculum (LES), the amount of guidance offered by the curriculum (HELP), whether or not loops or subroutines were required (REIT), the length of an economical correct solution (LNG), etc. For four of the six selected measures of problem difficulty, the linear models derived by the regressions were quite satisfactory, accounting for two-thirds or more of the variance. The best fit was for the proportion correct up to algebra, which we also felt was the best measure of programming difficulty per se; this model accounted for 85% of the variance. The two linear models that predicted the syntax error rate and the number of students who made semantic errors were less than satisfactory; both of these models accounted for less than half the variance. The independent variables entered into the regressions in different orders for all six regressions. However, on the average, it appeared that the most influential variables in predicting problem difficulty were (1) the position in the curriculum, (2) the expected proportion of conditional commands, (3) whether or not loops

or subroutines are required, and (4) the amount of guidance offered by the curriculum. The first and fourth of these can be characterized as curriculum-dependent variables, whereas the other two are problem-dependent.

We next looked more closely at the kinds of correct solutions produced by students. The 551 correct or nearly correct solutions given on first trials were classified according to four sets of criteria. Two methods of classification were based on the formal, or static, characteristics of the solutions, and two were based on functional, or dynamic, characteristics. These four methods of classification were referred to as _formal identity_, _formal equivalence_, _algorithmic equivalence_, and _functional equivalence_. We defined two programs to be formally identical only if the differences between them were such minor differences as the use of different letters for variables or the use of different part numbers for naming programs. Less trivial formal variations were allowed for formal equivalence. For algorithmic equivalence, we considered only the sequence of actions of a program and disregarded its form. The last equivalence relation, functional equivalence, was defined solely in terms of input and output; both the form of the program and the sequence of internal states were ignored.

The four equivalence relations exhibited considerable variance in their grouping power over the data. There were an average of eight types of solutions per problem when the solutions were classified by formal identity. Under formal equivalence, the average number of types was five. Under algorithmic equivalence, the average was 3.6, and under functional equivalence, 1.7.

Using the four equivalence relations discussed above, we defined four measures of diversity of solutions: diversity of function, diversity of algorithm, diversity of equivalent forms, and diversity of identical forms. The measure of diversity we used is akin to variance but, unlike variance, is appropriate for categorical scales of measurement. This measure--or rather, its unbiased estimator--is given by the formula

$$\hat{d} = 1 - \sum_{i=1}^{k} \frac{n_i(n_i-1)}{N(N-1)}$$

where N is the total number of solutions, n_i is the number of solutions in the i-th equivalence class (or, of the i-th type), and k is the number of equivalence classes (or types). As this measure of diversity is not widely known in psychology, we have included a precise mathematical discussion of its derivation and properties in Appendix B.

Once again we used the tool of the step-wise multiple linear regression to define predictive models for diversity. For this, we used the same set of 10 independent variables used in the prediction of problem difficulty. All four of the models accounted for more than 50% of the variance in diversity; the best fit was for diversity of function, in which 80% of the variance was accounted for. In examining the order in which the independent variables entered into the regressions we concluded that the three most important variables for the prediction of diversity are (1) whether loops or subroutines are required, (2) the expected proportion of conditional commands, and (3) the expected length of the solution. All three of these variables might be characterized as problem variables rather than curriculum variables. This contrast to our findings for the prediction of problem difficulty, in which two of the four most important variables were curriculum-dependent, leads us to conclude that problem difficulty could be more easily manipulated by the curriculum designer than could diversity. Another comparison of interest between the two sets of predictive models is that there are two independent variables that contribute largely to both. These two variables are (1) whether loops or subroutines are required and (2) the expected proportion of conditional commands. In view of the importance of conditionals, loops, and subroutines in programming, this is an intuitively satisfying result.

References

Davis, M. Computability & unsolvability. New York: McGraw-Hill, 1958.

Friend, J. E., Fletcher, J. D., and Atkinson, R. C. Student performance in computer-assisted instruction. Technical Report No. 184, Institute for Mathematical Studies in the Social Sciences, Stanford University, May 10, 1972.

Friend, J. E. Supplementary handbook for introduction to AID programming. Institute for Mathematical Studies in the Social Sciences, Stanford University, 1972.

Friend, J. E. Computer-assisted instruction in programming: A curriculum description. Technical Report No. 211, Institute for Mathematical Studies in the Social Sciences, Stanford University, July 31, 1973.

Friend, J. E. 100 programming problems. Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Kane, M. T. Variability in the proof behavior of college students in a CAI course in logic as a function of problem characteristics. Technical Report No. 192, Institute for Mathematical Studies in the Social Sciences, Stanford University, October 6, 1972.

Maloney, J. M. An investigation of college student performance on a logic curriculum in a computer-assisted instruction setting. Technical Report No. 183, Institute for Mathematical Studies in the Social Sciences, Stanford University, January 28, 1972.

APPENDIX A

The Programming Language AID

The subset of AID that is described herein includes that part of the language that is taught in the course "Introduction to Programming: AID." The following description is an excerpt from "100 Programming Problems."*

*Friend, J. E. 100 Programming Problems (with a description of the programming language AID). Institute for Mathematical Studies in the Social Sciences, Stanford University, September 1973.

AID Commands and Programs

Numbers and Algebraic Expressions

Algebraic expressions in the programming language AID follow ordinary algebraic notation quite closely. The letters A, B, C,..., Z are used as variables, and the following symbols are used for arithmetic operations and grouping:

+	addition
-	subtraction
*	multiplication
/	division
↑	exponentiation
! !	absolute value
( )	parentheses

In forming algebraic expressions, juxtaposition cannot be used to indicate multiplication; the expressions 2x and xy must be written as 2*X and X*Y in AID notation. Algebraic expressions must be given as a linear string of symbols, which precludes the use of the horizontal bar as indicator of division; $\frac{a}{b}$ must be written as A/B, and $\frac{a+b}{a-b}$ must be written as (A+B)/(A-B). Neither can subscripts or superscripts be used; x_i is written as X(I) and y^2 is written as Y↑2.

Grouping is indicated with parentheses just as in ordinary algebraic notations, and parentheses may be imbedded as desired. If parentheses are not used, arithmetic operations are performed in this order:

↑	
* and /	from left to right
+ and -	from left to right

Thus, exponentiation is always done first (unless parentheses are used to indicate otherwise), then either * or /, and finally either + or -. If two operations with the same order of precedence appear, they are evaluated in left-to-right order; in the expression X/Y*Z/W, the first operation to be performed will be X/Y.

AID numbers may be written in integer form (275) or in decimal form (5.87, 0.01, .72). Numbers are limited to nine significant digits and must be less than 10^{100} in absolute value. Numbers may also be written in a form of scientific notation that is a direct translation of ordinary scientific notation. For example, 2.3076×10^5 is written as 2.3076 * 10↑5. Since the slash (/) is used to indicate division, an expression like 2/3 is read as "two divided by three" rather than "two thirds." Because of this, an expression like X↑2/3 means $x^2 \div 3$, <u>not</u> $x^{2/3}$; to write $x^{2/3}$ in AID notation, use X↑(2/3).

Negative numbers are indicated by a minus sign: -2.7. When negative numbers are used in certain combinations, such as 2 + (-3), the negative number must be enclosed in parentheses; to be on the safe side, always use parentheses around negative numbers.

The variables A, B, C,..., Z may be used for numbers, as indicated above. They may also be used as indexed (subscripted) variables to identify lists of numbers or arrays of numbers. The list $x_1, x_2, \ldots, x_n$ is written in AID notation as X(1), X(2),..., X(N), and the entire list is then referred to simply as X. A two-dimensional array (matrix) of numbers may be identified by a variable using two indices: a_{ij} is written in AID as A(I,J). Up to 10 indices may be used (for up to 10-dimensional arrays). Indices may be given as numbers, or variables, or algebraic expressions; X(12), X(N,J), and

X(2*I+3, J/4). Regardless of how the indices are indicated they must have integer values and are limited to -250 to 250, including zero. Thus, the longest list of numbers has 501 members: X(-250), X(-249),..., X(-1), X(0), X(1),..., X(249), X(250). A two-dimensional array could have 501 × 501 members, etc.

To summarize, here are some examples of algebraic expressions and their AID equivalents:

$5x^2 + 3y^4$	5*X↑2 + 3*Y↑4
$z^{1/2}$	Z↑(1/2) or Z↑0.5
$\lvert x-y \rvert$	!x-y!
$x_1 + x_2 - x_3$	X(1) + X(2) - X(3)
A_{ij}	A(I,J)
$\frac{a+b+c+d}{4}$	(A+B+C+D)/4

In general, spaces may be used whenever desired in algebraic expressions. The expression 5*X+4 may also be written 5*X + 4 or 5 * X + 4 or 5* X+4. The exceptions to this rule are in indexed variables and, as we shall see later, in function notation. Expressions like X(5) and A(1,2) must be written without a space between the identifier and the opening parenthesis; X (5) or A (1,2) will cause an error message.

The Form of AID Commands

AID commands are quite similar to English commands:

```
TYPE X
SET Y = 7
STOP
```

Each command begins with a verb (TYPE, SET, STOP) and the form of the rest of the command depends upon the verb that is used. The verb TYPE, for example, may be followed by any algebraic expression (and the result will be that the expression is evaluated and the value typed on the user's tele-typewriter):

```
TYPE X - 2
TYPE 3/(4+2/7)
TYPE X↑2 + Y↑2
```

Some commands, like STOP, may consist of only one word, but most commands have either variables or algebraic expressions or equations or other kinds of arguments following the verb. Some commands also have optional modifiers, which are phrases that can be added to the command to modify its meaning. For example, the TYPE command may be modified by an IN FORM phrase:

```
TYPE X IN FORM 12
```

where Form 12 specifies the form in which X is to be typed. (This will be explained more fully below.)

With one exception (FORM), AID commands must be given in one line; a line is terminated by the user by typing the return key on the teletypewriter.

There are two kinds of AID commands: direct commands and indirect commands. Direct commands will be executed as soon as they are given, whereas indirect commands are stored and will not be executed until the user gives an order to do so. Many AID commands may be used as either direct or indirect commands. To indicate whether a command is to be a direct command or an indirect command, "step numbers" are used before indirect commands:

```
        12.7 TYPE 15/16 + 1/32
```

This command will be stored rather than executed immediately, and the step number may be used in later references to the command. When the user wishes to have the command executed, he gives a DO command like the following:

```
        DO STEP 12.7
```

Step numbers are decimal numbers between 1 and 10^9, and, like all numbers, are limited to 9 significant digits.

When indirect commands are stored, they are grouped into "parts" according to the integer portion of the step number. Commands numbered 23.2, 23.7, 23.84, and 23.001 are all grouped together into "Part 23." Indirect commands may be executed singly:

```
        DO STEP 23.2
```

or they may be executed in groups:

```
        DO PART 23
```

When the above command is given, all the steps in Part 23 will be executed, in numeric order. When Part 23 is exhausted, the execution will cease; even if there are steps numbered 24.1, 24.2, etc., execution will not automatically proceed to Part 24. A set of stored commands, to be executed as a group, is called a "program." A program may consist of a single part or, by the use of branching commands as explained below, several parts.

Although most AID commands can be used either as direct commands or indirect commands, there are a few that may be used only in one form. Table 1 lists the AID commands and shows which can be used directly and which indirectly.

Table 1

Direct and Indirect AID Commands

Command	May be used directly	May be used indirectly
DELETE	Yes	Yes*
DEMAND	No	Yes
DISCARD	Yes	Yes*
DO	Yes	Yes
FILE	Yes	Yes*
FORM	Yes	Yes
GO	Yes	No
LET	Yes	Yes
RECALL	Yes	Yes*
SET	Yes	Yes
SET (short version)	Yes	No
STOP	No	Yes
TO	No	Yes
TYPE	Yes	Yes
USE	Yes	Yes*

*Rarely used in the indirect form.

Basic Commands: SET, TYPE, DEMAND, TO, and DO

The five commands SET, TYPE, DEMAND, TO, and DO form the core of a basic AID vocabulary. Together with the algebraic expressions described above, a few standard AID functions, and the conditional clause described in the next section, these five commands are sufficient to solve any of the 100 problems given in this booklet.

The SET command is used to assign a value to a variable:

```
SET X = 12.7

SET K = 0.002305

SET M = K*X↑2
```

The algebraic expression used on the right of the equal sign may contain one or more other variables, but all of the variables used must have values so that the expression can be immediately evaluated. When a SET command is executed, the expression on the right of the equal sign is evaluated and that number is stored in temporary (core) storage with the specified identifier (the variable used on the left of the equal sign); that stored number may thereafter be referred to by its identifier. A SET command may be used to "define a variable in terms of itself." The result of the following sequence of commands would be that the number 7 is stored as N:

```
SET N = 13            sets N equal to 13.

SET N = N + 1         adds 1 to the current value of N.

SET N = N/2           divides the current value of N by 2.
```

SET may be used either indirectly (with a step number) or directly. If used as a direct command, the short form which omits the word SET may be used:

```
X = 7    equivalent to    SET X = 7

K = 0.07835    equivalent to    SET K = 0.078305
```

SET may also be used with indexed variables:

SET X(2,3) = 7	sets the element $X_{2,3}$ from the array X equal to 7
L(5) = 72.31	sets L_5 equal to 72.31

The TYPE command is used with an algebraic expression:

```
TYPE (X+K*Y)/3
```

Here again the algebraic expression must contain only variables that have values (or will be given values before the TYPE command is executed). When a TYPE command is executed, the value of the algebraic expression will be calculated and typed on the user's teletypewriter.

A TYPE command can be given with several arguments, separated by commas:

```
TYPE X,Y, (X+Y)/2
```

This command is equivalent to the three commands:

```
TYPE X

TYPE Y

TYPE (X+Y)/2
```

Caution: Only two commands, TYPE and DELETE, allow multiple arguments; other commands, like SET and DO, use only one argument.

The TYPE command can be used to type text by giving the text enclosed in quotation marks:

```
TYPE "TITLE: COMPOUND INTEREST CALCULATIONS"
```

Other uses of the TYPE command will be described later.

The DEMAND command can only be used indirectly (as a stored command):

```
20.4   DEMAND X
```

The DEMAND command uses a single variable as an argument, and the result of such a command is to cause the program to halt, type

```
X=
```

wait for the user to type a value for X, and then continue the execution of the program. By using DEMAND commands, a program can be written so as to ask for the data it needs. A useful variant of the DEMAND command is formed by appending the modifying phrase AS "text." The command

```
17.9 DEMAND R AS "INTEREST RATE"
```

will cause the program to stop at Step 17.9, type

```
INTEREST RATE=
```

and wait for the user to type a value which will be assigned the identifier R.

A feature of the DEMAND command that is frequently useful in iterated programs is that if the user refuses to give a value for the DEMANDed variable, and responds simply by typing the return key, the execution of the program will halt at that point; thus, seemingly endless loops can be used if they incorporate DEMANDs.

DEMAND is used solely for input, SET is used for both input and for internal computations, and TYPE is used for both computation and output. Here is an example of a complete program using all three of these commands:

```
4.1 TYPE "COMPUTATION OF INTEREST AT 4.5%"
4.2 SET R = 0.045
4.3 DEMAND P AS "PRINCIPAL"
4.4 SET I = R * P
4.5 SET T = P + I
4.6 TYPE I,T
```

This program would be executed by the command

```
        DO PART 4
```

and it would start by typing

```
        COMPUTATION OF INTEREST AT 4.5%
        PRINCIPAL =
```

As soon as the user typed a value for P, say 200, the program would reply

```
        I = 9
        T = 209
```

As mentioned, the steps within a part are ordinarily executed in numeric order. This order can be overridden by the use of the branching command, TO. TO, like DEMAND, can be used only as an indirect command. A TO command may be used to branch to either another step (within the same part or in some other part) or to another part:

6.3 TO STEP 7.29	will cause execution of Part 6 to cease and execution of Part 7 to commence at Step 7.29.
16.42 TO PART 8	will cause execution of Part 16 to cease and execution of Part 8 to commence at the lowest numbered step.

Although a TO command may be used unconditionally, as shown above, simply to alter the linear sequence of execution, it is more often used conditionally, that is, with an IF clause, as will be explained in the next section.

Several examples of direct DO commands have been given above. Used directly, DO causes the execution of a specified step or part:

```
        DO STEP 7.35
        DO PART 84
```

DO may also be used indirectly, as part of a program, to cause the execution of another part as a subroutine:

```
7.1  SET P = 3.14159
7.2  SET R = 15
7.3  DO PART 12
7.4  TYPE D, C, A
```

In this program Step 7.3 calls for the execution of Part 12. Part 12 is the "subroutine" and the DO command in Step 7.3 is the "subroutine call." When Part 7 is executed, the sequence of execution is:

Step 7.1

Step 7.2

Step 7.3

All of Part 12

Step 7.4

Thus, DO as well as TO can be used to override the automatic linear sequence of execution. The primary difference is that DO calls for another step or part to be inserted into the part being executed, whereas TO calls for a complete transfer of control to the part specified. Here are four sample commands, with comments, to summarize the difference between DO and TO.

3.6 DO PART 7	will cause all of Part 7 to be executed, followed by the execution of the remainder of Part 3.
3.6 TO PART 7	will cause all of Part 7 to be executed. Execution will halt at the end of Part 7. The remainder of Part 3 will not be executed automatically.
3.6 DO STEP 7.5	will cause Step 7.5 to be inserted as a one-step subroutine. After Step 7.5 is done, the remainder of Part 3 will be executed. No other steps in Part 7 will be done.
3.6 TO STEP 7.5	will cause execution of Part 7 to start at Step 7.5. Execution will halt at the end of Part 7, and the remainder of Part 3 will not be executed automatically.

There are two modifiers that may be used with DO commands: TIMES and FOR. The TIMES modifier is used to specify the number of times the required step or part will be executed:

```
DO STEP 3.5, 6 TIMES
13.2 DO PART 12, N TIMES
```

The number of times a step or part is to be iterated may be specified by a number or a variable, or even an algebraic expression, with the stipulation that the value is a positive integer.

The second modifier, the FOR clause, specifies values for some variable:

```
DO PART 4 FOR X = 7
```

This command is equivalent to the two commands

```
SET X = 7
DO PART 4
```

A list of values may be given in the FOR clause if desired:

```
DO PART 4 FOR X = 7, 23.8, 19
```

This command will cause Part 4 to be done three times, once for each of the listed values for X, and is thus equivalent to the six commands

```
SET X = 7
DO PART 4
SET X = 23.8
DO PART 4
SET X = 19
DO PART 4
```

The values for the variable may be given in the form of a "range specification," as in this example:

```
DO PART 21 FOR A = 5(2)13
```

The range specification 5(2)13 indicates that the initial value of A is to be 5 and that A is to be incremented by 2 with each successive iteration until the value of 13 is reached. That is, A will take on the values 5, 7, 9, 11, and 13. Any or all of the initial value, the size of the increment, and the final value may be given as algebraic expressions, and they need not be integral. The command

```
DO STEP 7.3 FOR Y = 3.2(.2)4
```

is equivalent to

```
DO STEP 7.3 FOR Y = 3.2, 3.4, 3.6, 3.8, 4
```

When values of a variable are given in a range specification, the final value is always used. Hence, the command

```
DO PART 2 FOR X = 0(2)7
```

will cause these values of X to be used: 0, 2, 4, 6, 7.

DO commands with either TIMES or FOR modifiers may, of course, be used as indirect steps to cause iterated execution of a subroutine.

<u>The IF Clause</u>

Certain modifiers, such as the AS or TIMES phrases, may be used to modify specific commands. There is one modifier that may be used with <u>any</u> AID command, and that is the IF clause. The addition of an IF clause changes any command from an "unconditional command" to a "conditional command." Here are a few examples:

```
TYPE X/Y IF Y > 0
3.2 DEMAND R IF T = A + X
7.3 DO PART 8, 3 TIMES IF X <= Y + 3
SET Z = X/(Q + S) IF Q + S > X
```

An IF clause contains the word IF followed by a Boolean expression. Boolean expressions (also called logical predicates) express relationships between numbers. The following relational symbols are used:

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
=	equal
#	not equal

As in ordinary usage, any algebraic expressions may be used in Boolean expressions:

```
X < 0
X + Y ↑ 2 # Z
2 > = Z
```

The Boolean operators AND, OR, and NOT may also be used:

```
NOT X < 0
X < 7  AND  Y > 8
X > 0  OR  X < Y - 2
X # 0  OR  Y # 0  OR  Z # 0
(A + B > 0  OR  A < 7)  AND  B > = 12
```

In evaluating Boolean expressions, the Boolean operators are evaluated in this order (unless there are parentheses to indicate otherwise):

NOT

AND

OR

When a conditional command is executed, the execution proceeds in two phases. First, the Boolean expression used in the IF clause is evaluated to determine whether it is true or false. Second, if the Boolean expression is true, the main clause will be executed.

Any command may be modified by an IF clause. One of the most important uses of the IF clause is in TO commands; a conditional TO command is called a "conditional branch" and is the principal mechanism used in writing non-linear programs, including those with loops. As an example, here is a simple program with a loop (this program simply counts from 0 to 30 by twos):

```
5.1  SET C = 0
5.2  TYPE C
5.3  SET C = C + 2
5.4  TO STEP 5.2 IF C < = 30
5.5  TYPE "THAT'S ALL."
```

Auxiliary Commands: FORM, LET, and DELETE

Besides the five commands (SET, TYPE, DEMAND, TO, and DO) that are used in writing simple programs, there are a number of auxiliary commands that are ordinarily used as direct commands. Two of these, FORM and LET, are to define forms and functions that will be used by TYPE and SET commands in programs, and are thus closely associated with the programs themselves. The other auxiliary commands are used more for bookkeeping or debugging purposes; these are DELETE, the file commands to be discussed in the following section, and the debugging commands to be discussed in the section after that.

FORM and LET are used in conjunction with stored programs. FORM is used to specify the format to be used for output. Ordinarily, when a TYPE command is used, the output is printed in a standard form. For example, when the command

```
TYPE (X + 2)/Y
```

is given, the value will be typed in this form:

```
(X + 2)/Y = 28.7
```

If a number is 10^6 or greater or if it is less than .001, it will be typed in scientific notation rather than decimal form:

```
(X + 2)/Y = 2.87 * 10↑(-4)
(X + 2)/Y = 2.87 * 10↑8
```

If the user prefers another form for output, he may specify it in a FORM statement. The FORM statement, unlike other AID commands, requires two lines; the first line specifies the form number (an integer between 1 and 10^9 to be used in later references) and the second line specifies the form itself:

```
FORM 12:
THE INTEREST IS ←←←.←←
```

The location of digits is indicated by the character ← and the position of the decimal point is shown by a period. When the form specified above is to be used, the TYPE command is modified by an IN FORM phrase:

```
TYPE P * R IN FORM 12
```

Numbers will be rounded to fit the specified form (which is the easiest way of rounding numbers to a fixed number of decimal places) and if no decimal point is specified, the number will be rounded to the nearest integer. When specifying a form, care must be taken to allow for as many digits before the decimal point as will be necessary; if an attempt is made to type a number in a form that is not large enough, an error message will result. If the number to be typed in a given form is negative, one of the digit locations will be taken up by the negative sign.

Any symbols, including punctuation marks, may be used in the text of a form:

```
FORM 42:
PRINCIPAL + INTEREST = $ ←←←.←←
```

No text is necessary if the user wishes merely to print a number in a given form and location.

More than one number may be provided for, and this is the only way in which more than one number can be printed on the same line:

```
FORM 6:
$ ←←←←.←← WILL EARN $ ←←←.←← INTEREST
```

To use a form with several numbers, the multiple-argument form of the TYPE command is required:

```
TYPE P, P * R IN FORM 6
```

The LET command is also used in conjunction with stored programs, but may be used independently for direct computations. The primary use of LET is in the definition of functions. The function $f(x) = 3x^2 + 2x$ is defined in AID as follows:

```
LET F(X) = 3*X↑2 + 2*X
```

When the function is used, in a SET or TYPE command, a value is substituted for the dummy variable X in the expression F(X):

```
SET Y = F(3)
TYPE F(5) - F(3.7)
```

The value that is substituted may be in the form of an algebraic expression, provided such an expression can be immediately evaluated:

```
SET N = 2
TYPE F(N/6)
```

Any of the variables A, B, C,..., Z may be used as function names. Take care, however, not to use the same identifier for both a real variable and a function since the first definition will be replaced by the second.

Functions of up to ten variables may be defined; here is an example of a function of three variables:

```
LET F(X, Y, Z) = (X*Y + Y*Z)/X*Y*Z
```

Caution: Do not use a space between the function name and the opening parentheses; an expression like F (3) will cause an error message.

A useful variant of the LET command is the conditional form of LET used to define functions conditionally. In ordinary notation, a function may sometimes be defined in this fashion:

$$f(x) = \begin{cases} -2x \text{ if } x < 0 \\ 5x \text{ if } x \geq 0 \end{cases}$$

In AID, this definition is given in a single line:

```
LET F(X) = (X < 0: -2*X; X > = 0: 5*X)
```

which is read "If $x < 0$, $f(x) = -2x$; if $x \geq 0$, $f(x) = 5x$." In the AID definition, the entire expression is enclosed in parentheses, the clauses within the definition are separated by semicolons, and each clause is divided into a condition and an algebraic expression separated from one another by a colon. Any number of clauses may be used; in the above example, there are two clauses.

If the definition of a function is given in ordinary terms with an "otherwise" clause,

$$f(x) = \begin{cases} 0 \text{ if } x < 0 \\ 2x \text{ if } x > = 0 \text{ and } x < 7 \\ 5x \text{ otherwise} \end{cases}$$

the AID definition does not require a condition in the final clause:

```
LET F(X) = (X < 0: 0; X > = 0 AND X < 7: 2*X; 5*X)
```

In this example, the final clause consists only of the algebraic expression 5*X, which will be used whenever all of the conditions in preceding clauses fail.

When a function definition is used, it is scanned from left to right until a condition that holds is found. Because of this, it is frequently possible to simplify AID definitions. For example, the condition in the second clause of the above example could be simplified from X > = 0 AND X < 7 to X < 7:

```
LET F(X) = (X < 0: 0; X < 7: 2*X; 5*X)
```

A function may call itself; hence, a variant of the conditional definition is definition by recursion. Here, for example, is the AID recursive definition of the factorial function X!

LET F(X) = (X = 1: 1; X*F(X-1))

Both LET and FORM serve to store information in core storage. In the one case a function definition is stored and in the other the definition of an output form. SET and DEMAND also use core storage; both of these cause a number and its identifier to be stored. Stored commands (indirect steps) are also put into core storage, as clued by the step number preceding the command. In programming it is often necessary to inspect the information that is being held in core or to delete some items. The contents of core can be displayed by using TYPE commands and deleted by means of DELETE commands. Some example of such TYPE and DELETE commands are given here, with comments:

TYPE X	will print the value of X if X is a number or a list or array, or the definition of X if X is a function.
DELETE X	will delete either a number X or a function X.
TYPE X(3)	will print the value of X_3.
DELETE X(3)	will delete the single value X_3 from the list X.
TYPE FORM 3	will type the definition of Form 3.
DELETE FORM 3	will delete the definition of Form 3.
TYPE STEP 7.1	will print the stored command identified as Step 7.1.
DELETE STEP 7.1	will delete Step 7.1.
TYPE PART 29	will print all of the steps in Part 29 in numeric order.
DELETE PART 29	will delete all of the steps in Part 29.

TYPE ALL	will print the entire contents of core.
DELETE ALL	will delete everything in core storage.
TYPE ALL VALUES	will print all numbers, lists, and arrays.
TYPE ALL FORMULAS	will print all function definitions.
TYPE ALL STEPS	
TYPE ALL PARTS	
TYPE ALL FORMS	
DELETE ALL VALUES	
DELETE ALL FORMULAS	
DELETE ALL STEPS	
DELETE ALL PARTS	
DELETE ALL FORMS	

Both TYPE and DELETE may be used with several arguments, separated by commas:

```
TYPE X, STEP 3.7, F
DELETE STEP 3.7, PART 9, K, F
```

These are the only two AID commands that have multiple-argument forms.

File Commands: USE, FILE, RECALL, and DISCARD

Anything that is stored in core will be automatically deleted whenever the user signs off. Any or all of this information can be copied to more permanent storage space on the disk. To do this, the file commands USE, FILE, RECALL, and DISCARD are used. AID files are variable length disk files, identified by integers from 1 to 2750. The files need not be used in numeric order and the user specifies which file he wants to use by giving a command like

```
USE FILE 100
```

The file number is held in core until another USE command is given (or until the user signs off), and all subsequent FILE, RECALL, and DISCARD commands will refer to this file.

Each file is divided into "items," numbered from 1 to 25, and the user must specify the item when storing or retrieving information. Items need not be used in numeric order. To file an item, a command like

```
FILE PART 7 AS ITEM 3
```

is given. The user may file a form, a step, a part, a value, a function definition, or all of these, using commands similar to the TYPE and DELETE command shown just above. The entire contents of core may be stored as a single item by giving a command like

```
FILE ALL AS ITEM 17
```

When information is filed on the disk, the contents of core are not disturbed; a copy is made for transfer to the disk.

When the user wishes to retrieve information from the file, he uses a command like

```
RECALL ITEM 17
```

and when he wishes to discard an item from the file, he uses a command like

```
DISCARD ITEM 17
```

Debugging Commands

The commands STOP and GO are used primarily for debugging purposes. STOP is inserted as a temporary command, to be removed when debugging is complete, and may be used either conditionally or unconditionally to halt the execution of the program at the point where the STOP command is encountered:

```
47.3 STOP
47.352 STOP IF N > 100
```

While the program is STOPped, the user may inspect or alter the contents of core, checking current values of variables used by the program; replacing, inserting, or deleting steps in the program, etc. To resume execution the user gives the direct command

```
GO
```

During the time the program is STOPped, the user may not execute another step or part (that is, he cannot give another direct DO command), at least not if he wishes to resume the execution of the STOPped program at a later time.

GO may also be used to restart the execution of a program that was halted because of a syntax error. After the program stops, and the error message is printed, the user may correct the error and then resume execution from that point by giving a direct GO command.

Temporary TYPE commands may also be used for debugging purposes. These are commands like

```
32.105 TYPE X, Y, K, N
```

that are inserted temporarily so that the values of variables will be typed for inspection. When debugging is complete, these commands, and temporary STOP commands, are removed by giving DELETE commands:

```
DELETE STEP 47.3, STEP 32.105
```

Summary of AID Commands

The following summary of AID commands is given in the form of examples, with comments. Commands that are ordinarily used directly are shown without step numbers and those that are ordinarily used indirectly are shown with step numbers; to find out which commands must be used directly (or indirectly) refer to Table 1.

Most of the examples are shown as unconditional commands; however, any command may be used conditionally (modified by an IF clause) if desired.

DELETE X	deletes the identifier X and its value.
DELETE F	deletes the definition of the function F.
DELETE A(2,3)	deletes the element $A_{2,3}$ from the array A.
DELETE STEP 7.1	deletes Step 7.1.
DELETE PART 7	deletes all steps in Part 7.
DELETE FORM 22	deletes the definition of Form 22.
DELETE K, STEP 4.3, STEP 4.4	deletes the three specified items.
DELETE ALL VALUES	deletes all real variables and their values.
DELETE ALL STEPS	etc.
DELETE ALL PARTS	
DELETE ALL FORMS	
DELETE ALL	

7.1 DEMAND M	requests a value for the real variable M.
2.05 DEMAND A(2,3)	requests a value for the element $A_{2,3}$ of the array A.
3.7 DEMAND X(I,J,K)	requests a value for the element $X_{i,j,k}$ of the three-dimensional array X.
16.4 DEMAND X AS "RADIUS"	requests a value for X by typing RADIUS =

DISCARD ITEM 20	discards Item 20 from the previously designated disk file (see USE).
DO STEP 6.2	executes Step 6.2.
DO PART 9	executes the steps in Part 9 in numeric order.
DO PART 12, 7 TIMES	executes Part 12, 7 times.
DO PART 4 FOR X = 2, 7, 4.3	executes Part 4, 3 times, once with X = 2, once with X = 7, and once with X = 4.3.
7.2 DO PART 6, N TIMES	executes Part 6 (as a subroutine), N times.
62.15 DO STEP 32.3 FOR A = 5(2)12	executes Step 32.3 once for each of these values of A: 5, 7, 9, 11, 12.
FILE X AS ITEM 2	files the identifier X and its value as Item 2 of the previously designated disk file (see USE).
FILE A(7,3) AS ITEM 6	
FILE FORM 3 AS ITEM 12	
FILE STEP 6.25 AS ITEM 4	
FILE PART 9 AS ITEM 1	
FILE ALL STEPS AS ITEM 5	
FILE ALL PARTS AS ITEM 21	
FILE ALL FORMS AS ITEM 7	
FILE ALL VALUES AS ITEM 14	
FILE ALL AS ITEM 3	

(Note: The item number must be an integer from 1 to 25.)

FORM 7: THE LENGTH IS ←←← INCHES MORE THAN THE WIDTH.	defines an output form with allowance for one value (see TYPE...IN FORM...).
FORM 13: ←←.← ←←.← ←←.←	defines an output form with allowance for three values, but no text.
FORM 2: THE COST OF ←← ITEMS IS $ ←←←.←←	defines an output form with allowance for two values. The first value will be rounded to the nearest integer, and the second value will be rounded to two decimal places.

(Note: The form number must be a positive integer less than 10^9.)

GO	continues the execution of a program halted by a STOP command or by a syntax error.

LET F(X) = 3*X↑5 - 7	defines the function $f(x) = 3x^5 - 7$.
LET V(R,H) = 3.14159265*R↑2*H	defines the function $V(r,h) = \pi r^2 h$ (functions of up to 10 variables may be defined).
LET F(X) = (X < 0: X↑2 + 5; X >= 0: X + 5)	defines the function $f(x) = \begin{cases} x^2 + 5 \text{ if } x < 0 \\ x + 5 \text{ if } x \geq 0 \end{cases}$
LET F(X) = (X = 1: 1; X + F(X-1))	defines the recursive function $f(x) = \begin{cases} 1 \text{ if } x = 1 \\ x + f(x-1) \text{ if } x > 1 \end{cases}$

RECALL ITEM 7	recalls Item 7 from the previous designated disk file (see USE).
SET P = 3.14159265	assigns the value 3.14159265 to the identifier P.
6.35 SET A(5, 7) = 12.31	assigns the value 12.31 to the element $A_{5,7}$ in the array A.
7.3 SET N = N + 1	increases the current value of N by 1.
X = 4.3	short form of the SET command, equivalent to SET X = 4.3
L(7) = 2769	short form of the SET command, equivalent to SET L(7) = 2769
7.3 STOP	causes the program to stop execution of Step 7.3 (see GO).
26.64 STOP IF N > M + 1	causes the execution of the program to stop at Step 26.64 if N > M + 1.
31.3 TO STEP 31.1 IF N < 100	causes a branch to Step 31.1 if N < 100
8.25 TO PART 9	causes an unconditional branch to Part 9.
TYPE X↑Y	evaluates x^y and types the result.
7.3 TYPE X, F(X)	types the values of X and F(X).
12.9 TYPE "TAX COMPUTATIONS"	types an exact copy of the text enclosed in quotation marks.
TYPE FORM 2	types the definition of Form 2.
TYPE STEP 3.7	types the command stored as Step 3.7.
TYPE PART 5	types all of the commands in Part 5.

TYPE ALL STEPS	
TYPE ALL PARTS	
TYPE ALL.FORMS	
TYPE ALL VALUES	
TYPE ALL	
3.8 TYPE 5*X IN FORM 2	evaluates 5x and types the result in the specified output form (see FORM).
USE FILE 100	designates the disk file to be used by subsequent FILE, RECALL, and DISCARD commands.

(Note: The file number must be a positive integer from 1 to 2750.)

AID Functions

In addition to the functions that may be defined by the user by means of LET commands, there are a number of useful standard AID functions. There are two trigonometric functions, SIN(X) and COS(X); X is in radians and must have an absolute value less than 100. The natural logarithm function LOG(X) yields the logarithm to the base e of x, where x is any positive real number. The inverse of the LOG function is the exponential function EXP(X), equivalent to e^x.

Several functions depend upon features of the decimal representation or scientific notation of the argument:

IP(X), the "integer part" function, yields the integer portion of the decimal representation of the number x. For example, IP(7304.56) = 7304.

FP(X), the "fraction part" function, yields the fractional portion of the decimal representation of the number x. FP(7304.56) = .56.

DP(X), the "digit part" function, yields the digital part of the scientific notation of x. For example, DP(3789.54) = 3.78954 since the scientific notation for x is 3.78954×10^3.

XP(X), the "exponent part" function, yields the exponent part of the scientific notation. For example, XP(3789.54) = 3 since 3 is used as the exponent of 10 in the representation 3.78954×10^3.

Two other real functions that are occasionally used are SGN(X), the "sign" function, and SQRT(X), the "square root" function. These are defined as follows:

$$\mathrm{SGN}(X) = \begin{cases} 1 \text{ if } x \text{ is positive} \\ 0 \text{ if } x \text{ is zero} \\ -1 \text{ if } x \text{ is negative} \end{cases}$$

$$\mathrm{SQRT}(X) = \sqrt{X}$$

There are four functions on lists of real numbers: MAX, MIN, SUM, and PROD. The forms of these are similar, and the resulting values are, respectively, the maximum of the specified list, the minimum, the sum of the numbers in the list, and the product. Each of these four functions may be used by simply listing the members of the argument:

MIN(.69, 2/3, .63) has a value of .63

SUM(2, 15, 0, 4) has a value of 21

The list of numbers to be used as an argument may be given by specifying a formula and the values of the dummy variable used in the formula:

SUM(I = 2, 10, 3: I * 5) is equivalent to

SUM(2 * 5, 10 * 5, 3 * 5).

The values of the variable may be given in a range specification:

SUM(I = 5(1)10: 3/I-7)

This expression is equivalent to

$$\sum_{i=5}^{10} \left(\frac{3}{i} - 7\right)$$

Similarly, the expression

PROD(J = 0(2)6: J↑2)

is equivalent to

$$\prod_{j=0,2,4,6} (j^2)$$

The function FIRST is a function on an indexed list of Boolean expressions. For a specified list of Boolean expressions, the FIRST function will yield the index of the first true expression. That is, it will find the location of the first true predicate. The form of the FIRST function is shown in this example:

FIRST(I = 1(1)50: I > 6↑2 + 3)

The value of this expression will be the first value of i in the set $\{1, 2, 3, \ldots, 50\}$ such that $i > 6^2 + 3$ (that value is 40).

Another simpler function on Boolean expressions is the function TV(X) which yields either 1 or 0 depending upon whether the Boolean expression X is true or false. For example, the value of TV(2 < 1 OR 5 > 4) is 1.

For all of the standard AID functions, the values are real numbers; hence, these functions can be used anywhere in algebraic expressions just as in ordinary algebraic notation. They may also be combined and composed in the usual ways. Here are a few examples of algebraic expressions in ordinary notation and in AID notation:

$\frac{\sin x}{\cos x}$	SIN(X)/COS(X)
$\sin^2 x$	(SIN(X))↑2
$\ln x$	LOG(X)
e^{2x}	EXP(2*X)
$\sqrt{x^2 + y^2}$	SQRT(X↑2 + Y↑2)

APPENDIX B

The Forms of Programs Written by Students (with Division into Equivalence Classes using Four Definitions of Program Equivalence)

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...: Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L5-30	SET K = .3937 TYPE 6.9/K, 7.445/K, 23.9753/K	19	I_1	E_1	A_1	F_1

	SET K = .3937
	SET A = 6.9
	SET B = 7.445
	SET C = 23.9753
	TYPE A/K, B/K, C/K
		6	I_2	E_2	A_2	F_1

	SET K = .3937 TYPE 6.9/K TYPE 7.445/K, 23.9753/K	2	I_3	E_1	A_1	F_1
	SET K = 1/.3937 TYPE 6.9*K, 7.445*K, 23.9753*K	2	I_4	E_3	A_3	F_1

	SET K = .3937
	SET Y = 6.9
	TYPE Y/K
	SET Y = 7.445
	TYPE Y/K
	SET Y = 23.9753
	TYPE Y/K
		2	I_5	E_4	A_4	F_1

	SET K = .3937 TYPE 6.9/K TYPE 7.445/K TYPE 23.9753/K	1	I_6	E_1	A_1	F_1
	TYPE 6.9/.3937, 7.445/.3937, 23.9753/.3937	1	I_7	E_5	A_5	F_1
		N=33	7 classes	5 classes	5 classes	1 class

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L8-9	LET F(X) = 1/X TYPE F(119.4), F(67.3↑3), F(6+4)	18	I_1	E_1	A_1	F_1
	LET F(X) = 1/X TYPE F(119.4) TYPE F(67.3↑3) TYPE F(6+4)	15	I_2	E_1	A_1	F_1
	LET F(X) = X↑(-1) TYPE F(119.4) TYPE F(67.3↑3) TYPE F(6+4)	2	I_3	E_1	A_1	F_1
	LET F(X) = X↑(-1) TYPE F(119.4), F(67.3↑3), F(6+4)	1	I_4	E_1	A_1	F_1
		N=36	4 classes	1 class	1 class	1 class
L8-27	LET V(R,H) = 3.1416*R↑2*H TYPE V(18.6, 57.5), V(19.3, 65.4)	7	I_1	E_1	A_1	F_1
	LET V(R,H) = 3.1416*R↑2*H TYPE V(18.6, 57.5) TYPE V(19.3, 65.4)	5	I_2	E_1	A_1	F_1

	LET V(R,H) = 3.1416*R↑2*H
	SET R = 18.6
	SET H = 57.5
	TYPE V(R,H)
	SET R = 19.3
	SET H = 65.4
	TYPE V(R,H)
		4	I_3	E_2	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L8-27 (cont.)	LET V(H,R) = 3.1416*R↑2*H TYPE V(57.5, 18.6) TYPE V(65.4, 19.3)	3	I_4	E_3	A_1	F_1
	LET V(R,H) = R↑2*H*3.1416 TYPE V(18.6, 57.5), V(19.3, 65.4)	2	I_5	E_1	A_1	F_1
	LET V(X,Y,Z) = X*Y↑2*Z SET X = 3.1416 TYPE V(X, 18.6, 57.5), V(X, 19.3, 65.4)	2	I_6	E_4	A_4	F_1
	LET V(R,H) = (R↑2)*H*3.1416 TYPE V(18.6, 57.5), V(19.3, 65.4)	1	I_7	E_1	A_1	F_1
	LET V(H,R) = 3.1416*R↑2*H TYPE V(57.5, 18.6), V(65.4, 19.3)	1	I_8	E_3	A_1	F_1
	LET V(H,R) = 3.1416*H*R↑2 TYPE V(57.5, 18.6) TYPE V(65.4, 19.3)	1	I_9	E_3	A_1	F_1

	LET V(R,H) = 3.1416*R↑2*H
	SET H = 57.5
	SET R = 18.6
	TYPE V(R,H)
	SET H = 65.4
	SET R = 19.3
	TYPE V(R,H)
		1	I_{10}	E_2	A_3	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence

L8-27 (cont.)	LET V(H,R) = 3.1416*R↑2*H
	SET H = 57.5
	SET R = 18.6
	TYPE V(H,R)
	SET H = 65.4
	SET R = 19.3
	TYPE V(H,R)
		1	I_{11}	E_5	A_3	F_1

	LET V(H,R) = 3.1416*H*R*R
	SET H = 57.5
	SET R = 18.6
	TYPE V(H,R)
	SET H = 65.4
	SET R = 19.3
	TYPE V(H,R)
		1	I_{12}	E_5	A_3	F_1

		N=29	12 classes	5 classes	4 classes	1 class
L8-28	LET C(F) = (F - 32)*5/9 TYPE C(0), C(10), C(32), C(72), C(212)	8	I_1	E_1	A_1	F_1
	LET C(F) = 5/9*(F - 32) TYPE C(0), C(10), C(32), C(72), C(212)	3	I_2	E_1	A_1	F_1
	LET C(F) = (F - 32)*5/9 TYPE C(0) TYPE C(10) TYPE C(32) TYPE C(72) TYPE C(212)	3	I_3	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L8-28 (cont.)	LET C(F) = (F - 32)*(5/9) TYPE C(0) TYPE C(10), C(32), C(72), C(212)	2	I_4	E_1	A_1	F_1
	LET C(F) = (F - 32)*(5/9) TYPE C(0), C(10), C(32), C(72), C(212)	2	I_5	E_1	A_1	F_1
	LET C(F) = (F - 32)*5/9 SET F = 0 TYPE C(F) SET F = 10 TYPE C(F) SET F = 32 TYPE C(F) SET F = 72 TYPE C(F) SET F = 212 TYPE C(F)	2	I_6	E_2	A_2	F_1
	LET C(F) = 5/9*(F - 32) TYPE C(0) TYPE C(10) TYPE C(32) TYPE C(72) TYPE C(212)	1	I_7	E_1	A_1	F_1
	LET C(F) = (F - 32)*(5/9) TYPE C(0) TYPE C(10) TYPE C(32) TYPE C(72) TYPE C(212)	1	I_8	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L8-28 (cont.)	LET C(F) = (F - 32)*5/9 TYPE C(0) TYPE C(10) TYPE C(32), C(72), C(212)	1	I_9	E_1	A_1	F_1
	LET C(F) = (5/9)*(F - 32) SET F = 0 TYPE C(F) SET F = 10 TYPE C(F) SET F = 32 TYPE C(F) SET F = 72 TYPE C(F) SET F = 212 TYPE C(F)	1	I_{10}	E_2	A_2	F_1
	LET C(F) = 5/9*(F - 32) SET F = 0 TYPE C(F) SET F = 10 TYPE C(F) SET F = 32 TYPE C(F) SET F = 72 TYPE C(F) SET F = 212 TYPE C(F)	1	I_{11}	E_2	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L8-28 (cont.)	LET C(F) = (F - 32)/1.8 TYPE C(0) TYPE C(10) TYPE C(32) TYPE C(72) TYPE C(212)	1	I_{12}	E_1	A_1	F_1
		N=26	12 classes	2 classes	2 classes	1 class
L9-3	LET H(A,B) = SQRT(A↑2 + B↑2) TYPE H(3,4), H(12,12), H(1/2,3/4), H(9,13.2)	6	I_1	E_1	A_1	F_1
	LET H(A, B) = SQRT(A↑2 + B↑2) TYPE H(3, 4) TYPE H(12, 12) TYPE H(1/2, 3/4) TYPE H(9,13.2)	5	I_2	E_1	A_1	F_1
	LET H(A, B) = SQRT(A↑2 + B↑2) TYPE H(3, 4) TYPE H(12, 12) TYPE H(.5, .75) TYPE H(9, 13.2)	3	I_3	E_1	A_1	F_1
	LET H(A, B) = (A↑2 + B↑2)↑(1/2) TYPE H(3, 4) TYPE H(12, 12) TYPE H(1/2, 3/4) TYPE H(9, 13.2)	3	I_4	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L9-3 (cont.)	LET H(A, B) = SQRT(A↑2 + B↑2) TYPE H(3,4) TYPE H(12, 12), H(1/2, 3/4), H(9, 13.2)	2	I_5	E_1	A_1	F_1
	LET H(A, B) = (A↑2 + B↑2)↑(1/2) TYPE H(3, 4), H(12, 12), H(1/2, 3/4), H(9, 13.2)	1	I_6	E_1	A_1	F_1

	LET H(A, B) = (A↑2 + B↑2)↑(1/2)
	SET C = 3
	TYPE H(C, 4), H(12, 12), H(1/2, 3/4),
	H(9, 13.2)
		1	I_7	E_3	A_3	F_1

	LET H(A, B) = SQRT(A↑2 + B↑2)
	SET A = 3
	SET B = 4
	TYPE H(A, B)
	SET A = 12
	SET B = 12
	TYPE H(A, B)
	SET A = 1/2
	SET B = 3/4
	TYPE H(A, B)
	SET A = 9
	SET B = 13.2
	TYPE H(A, B)
		1	I_8	E_2	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence

L9-3 (cont.)	LET H(A, B) = (A↑2 + B↑2)↑(1/2)
	SET A = 3
	SET B = 4
	TYPE H(A, B)
	SET A = 12
	SET B = 12
	TYPE H(A, B)
	SET A = 1/2
	SET B = 3/4
	TYPE H(A, B)
	SET A = 9
	SET B = 13.2
	TYPE H(A, B)
		1	I_9	E_2	A_2	F_1

		N=23	9 classes	3 classes	3 classes	1 class
L9-8	LET Q(M, N) = IP(M/N) TYPE Q(9172), 38) TYPE Q(13, 87) TYPE Q(768, 101) TYPE Q(6480, 15)	11	I_1	E_1	A_1	F_1
	LET Q(M, N) = IP(M/N) TYPE Q(9172, 38), Q(13, 87), Q(768, 101), Q(6480, 15)	8	I_2	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence

L9-8 (cont.)	LET Q(M, N) = IP(M/N)
	SET M = 9172
	SET N = 38
	TYPE Q(M, N)
	SET M = 13
	SET N = 87
	TYPE Q(M, N)
	SET M = 768
	SET N = 101
	TYPE Q(M, N)
	SET M = 6480
	SET N = 15
	TYPE Q(M, N)
		3	I_3	E_2	A_2	F_1

	LET Q(M, N) = IP(M/N) TYPE Q(9172, 38) TYPE Q(13, 87), Q(768, 101), Q(6480, 15)	2	I_4	E_1	A_1	F_1
	LET Q(M, N) = IP(M/N) TYPE Q, Q(9172,38), Q(13,87), Q(768,101), Q(6480,15)	1	I_5	E_3	A_3	F_2

	LET Q(X) = IP(X)
	SET A = 9172/38
	SET B = 13/87
	SET C = 768/101
	SET D = 6480/15
	TYPE Q(A), Q(B), Q(C), Q(D)
		1	I_6	E_4	A_4	F_1

		N=26	6 classes	4 classes	4 classes	2 classes

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L10-12	1.1 TYPE M*5280/(60*60) DO STEP 1.1 FOR M = 10, 100, 65, 1023	15	I_1	E_1	A_1	F_1
	1.1 TYPE M*5280/3600 DO STEP 1.1 FOR M = 10, 100, 65, 1023	4	I_2	E_1	A_1	F_1
	1.1 TYPE M*5280/(60↑2) DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_3	E_1	A_1	F_1
	1.1 TYPE 5280/3600*M DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_4	E_1	A_1	F_1
	1.1 TYPE 5280/3600*(M) DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_5	E_1	A_1	F_1
	1.1 TYPE M/60/60*5280 DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_6	E_1	A_1	F_1
	1.1 TYPE 5280*M*1/3600 DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_7	E_1	A_1	F_1
	1.1 TYPE M*(5280/3600) DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_8	E_1	A_1	F_1
	1.1 TYPE (M*5280)/3600 DO STEP 1.1 FOR M = 10, 100, 65, 1023	1	I_9	E_1	A_1	F_1
	1.1 LET F(M) = (M*5280)/3600 1.2 TYPE F(M) DO PART 1 FOR M = 10, 100, 65, 1023	1	I_{10}	E_2	A_1	F_1
		N=27	10 classes	2 classes	1 class	1 class

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L10-19	1.1 TYPE SQRT(X) DO STEP 1.1 FOR X = 1(1)10	18	I_1	E_1	A_1	F_1
	1.1 TYPE X↑(1/2) DO STEP 1.1 FOR X = 1(1)10	11	I_2	E_1	A_1	F_1
	1.1 TYPE X↑.5 DO STEP 1.1 FOR X = 1(1)10	1	I_3	E_1	A_1	F_1
		N=30	3 classes	1 class	1 class	1 class
L11-11	1.1 SET D = 2*R 1.2 SET C = D*3.1416 1.3 SET A = 3.1416*R↑2 1.4 TYPE R, D, C, A DO PART 1 FOR R = 10(10)50	6	I_1	E_1	A_1	F_1
	1.1 TYPE R 1.2 TYPE 2*R 1.3 TYPE 2*3.1416*R 1.4 TYPE 3.1416*R↑2 DO PART 1 FOR R = 10(10)50	5	I_2	E_2	A_2	F_1
	1.1 TYPE R 1.2 SET D = 2*R 1.3 TYPE D 1.4 SET C = 3.1416*D 1.5 TYPE C 1.6 SET A = 3.1416*R↑2 1.7 TYPE A DO PART 1 FOR R = 10(10)50	4	I_3	E_3	A_3	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L11-11 (cont.)	1.1 TYPE R 1.2 TYPE 2*R 1.3 TYPE 2*3.1416*R 1.4 TYPE 3.1416*R↑2 DO PART 1 FOR R = 10, 20, 30, 40, 50	2	I_4	E_2	A_2	F_1
	1.1 SET D = 2*R 1.2 SET P = 3.1416 1.3 SET C = D*P 1.4 SET A = P*R↑2 1.5 TYPE R, D, C, A DO PART 1 FOR R = 10, 20, 30, 40, 50	1	I_5	E_3	A_4	F_1
	1.1 SET D = 2*R 1.2 SET C = D*3.1416 1.3 SET A = 3.1416*R↑2 1.4 TYPE R, D, C, A DO PART 1 FOR R = 10, 20, 30, 40, 50	1	I_6	E_1	A_1	F_1
	1.1 LET D = 2*R 1.2 LET C = D*3.1416 1.3 LET A = 3.1416*R↑2 1.4 TYPE R, D, C, A DO PART 1 FOR R = 10(10)50	1	I_7	E_4	A_2	F_1
	1.1 SET D = 2*R 1.2 SET P = 3.1416 1.3 SET C = D*P 1.4 SET A = P*R↑2 1.5 TYPE R, D, C, A DO PART 1 FOR R = 10(10)50	1	I_8	E_3	A_4	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L11-11 (cont.)	1.1 SET P = 3.1416 1.2 SET D = 2*R 1.3 SET C = D*P 1.4 SET A = P*R↑2 1.5 TYPE R, D, C, A DO PART 1 FOR R = 10(10)50	1	I_9	E_3	A_5	F_1
	1.1 SET D = 2*R 1.2 SET C = 3.1416*D 1.3 SET A = 3.1416*R↑2 1.4 TYPE R 1.5 TYPE D 1.6 TYPE C 1.7 TYPE A DO PART 1 FOR R = 10(10)50	1	I_{10}	E_1	A_1	F_1
	1.1 SET D = 2*R 1.2 SET C = 3.1416*D 1.3 SET A = 3.1416*R↑2 1.4 TYPE R 1.5 TYPE D 1.6 TYPE C 1.7 TYPE A DO PART 1 FOR R = 10, 20, 30, 40, 50	1	I_{11}	E_1	A_1	F_1
	1.1 SET D = 2*R 1.2 SET C = 3.1416*D 1.3 SET A = 3.1416*R↑2 1.4 TYPE R 1.5 TYPE D, C, A DO PART 1 FOR R = 10, 20, 30, 40, 50	1	I_{12}	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L11-11 (cont.)	1.1 SET P = 3.1416 1.2 TYPE R, 2*R, 2*P*R, P*R 2 DO PART 1 FOR R = 10(10)50	1	I_{13}	E_5	A_6	F_1
		N=26	13 classes	5 classes	6 classes	1 class
L12-4	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 TYPE (A + B + C)/3 DO PART 1	32	I_1	E_1	A_1	F_1
	LET V(A,B,C) = (A + B + C)/3 1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 TYPE V(A, B, C) DO PART 1	1	I_2	E_2	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 SET N = 3 1.5 SET T = A + B + C 1.6 SET M = T/N 1.7 TYPE M DO PART 1	1	I_3	E_3	A_2	F_1
		N=34	3 classes	3 classes	2 classes	1 class

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L13-29	1.1 DEMAND Y 1.2 TYPE Y*12 DO PART 1, ... TIMES	11	I_1	E_1	A_1	F_1
	1.1 LET C(X) = 12*X 1.2 DEMAND X 1.3 TYPE C(X) DO PART 1, ... TIMES	5	I_2	E_2	A_1	F_1
	1.1 DEMAND Y 1.2 SET M = 12*Y 1.3 TYPE M DO PART 1, ... TIMES	3	I_3	E_3	A_2	F_1
	1.1 DEMAND Y 1.2 TYPE Y*12 DO PART 1	2	I_4	E_1	A_1	F_1
	1.1 DEMAND Y 1.2 LET M = 12*Y 1.3 TYPE M DO PART 1, ... TIMES	1	I_5	E_4	A_1	F_1
	1.1 DEMAND Y 1.2 SET M = Y*12 1.3 TYPE Y 1.4 TYPE M DO PART 1, ... TIMES	1	I_6	E_5	A_3	F_2

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L13-29 (cont.)	1.1 DEMAND X 1.2 TYPE X 1.3 TYPE X*12 DO PART 1, ... TIMES	1	I_7	E_6	A_4	F_2
	1.1 DEMAND Y AS "YEARS" 1.2 LET N(Y) = Y*12 1.3 TYPE N(Y) 1.4 TO STEP 1.1 DO PART 1	1	I_8	E_7	A_5	F_1
		N=25	8 classes	7 classes	5 classes	2 classes
L15-15	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF X >= Y DO PART 1	4	I_1	E_1	A_1	F_1
	1.1 TYPE X IF X < Y 1.2 TYPE Y IF Y <= X SET Y = ... DO PART 1 FOR X = ...	3	I_2	E_2	A_1	F_1
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X <= Y 1.4 TYPE Y IF X > Y DO PART 1	2	I_3	E_3	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-15 (cont.)	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF Y <= X DO PART 1, ... TIMES	1	I_4	E_7	A_1	F_1
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF X > Y DO PART 1	1	I_5	E_5	A_2	F_2
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF X > Y DO PART 1, ... TIMES	1	I_6	E_6	A_2	F_2
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF Y < X DO PART 1	1	I_7	E_5	A_2	F_2
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF Y < X DO PART 1, ... TIMES	1	I_8	E_6	A_2	F_2

Problem Number	Form of Program	N	Equivalence Class when-Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-15 (cont.)	1.1 TYPE X IF X <= Y 1.2 TYPE Y IF Y < X SET Y = ... DO PART 1 FOR X = ...	1	I_9	E_4	A_1	F_1
	1.1 TYPE X IF X <= Y 1.2 TYPE Y IF X > Y SET Y = ... DO PART 1 FOR X = ...	1	I_{10}	E_4	A_1	F_1
	1.1 TYPE X IF X < Y 1.2 TYPE Y IF Y < X SET Y = ... DO PART 1 FOR X = ...	1	I_{11}	E_8	A_2	F_2
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE Y IF X > Y 1.4 TYPE X IF X < Y DO PART 1	1	I_{12}	E_9	A_2	F_2
	1.1 DEMAND X 1.2 DEMAND Y 1.3 TYPE X IF X < Y 1.4 TYPE Y IF Y < X 1.5 TO STEP 1.1 DO PART 1	1	I_{13}	E_{10}	A_2	F_2
	1.1 TYPE X IF X < Y 1.2 TYPE Y IF Y < X SET X = ... SET Y = ... DO PART 1	1	I_{14}	E_{11}	A_2	F_2
		N=20	14 classes	11 classes	2 classes	2 classes

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-17	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 DEMAND D 1.5 SET S = A 1.6 SET S = B IF B < S 1.7 SET S = C IF C < S 1.8 SET S = D IF D < S 1.9 TYPE "THE SMALLEST NUMBER IS" 1.95 TYPE S DO PART 1	13	I_1	E_1	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 DEMAND D 1.5 SET S = A 1.6 SET S = B IF B < S 1.7 SET S = C IF C < S 1.8 SET S = D IF D < S 1.9 TYPE S DO PART 1	5	I_2	E_2	A_2	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 DEMAND D 1.5 SET S = A 1.6 SET S = B IF B < S 1.7 SET S = C IF C < S 1.8 SET S = D IF D < S 1.9 TYPE S DO PART 1, ... TIMES	1	I_3	E_3	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-17 (cont.)	1.1 DEMAND A 1.2 SET S = A 1.3 DEMAND B 1.4 SET S = B IF B < S 1.5 DEMAND C 1.6 SET S = C IF C < S 1.7 DEMAND D 1.8 SET S = D IF D < S. 1.9 TYPE S DO PART 1	1	I_4	E_2	A_3	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 DEMAND D 1.5 SET S = A 1.6 SET S = B IF B < S 1.7 SET S = C IF C < S 1.8 SET S = D IF D < S 1.9 TYPE "THE SMALLEST NUMBER IS" 1.91 TYPE S 1.92 TO STEP 1.1 DO PART 1	1	I_5	E_4	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 DEMAND D 1.5 DO PART 2 2.1 SET S = A 2.2 SET S = B IF B < A 2.3 SET S = C IF C < S 2.4 SET S = D IF D < S 2.5 TYPE S DO PART 1	1	I_6	E_5	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-17 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 DEMAND D 1.5 SET S = A 1.6 SET S = B IF B < S 1.7 SET S = C IF C < S 1.8 SET S = D IF D < S 1.9 TYPE "THE SMALLEST NUMBER IS" 1.91 TYPE S DO PART 1, ... TIMES	1	I_7	E_6	A_1	F_1
		N=23	7 classes	6 classes	3 classes	1 class
L15-18	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE A, B IF A > B 1.4 TYPE B, A IF B > A 1.5 TYPE A IF A = B DO PART 1, ... TIMES	6	I_1	E_2	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE A, B IF A > B 1.4 TYPE B, A IF A < B 1.5 TYPE A IF A = B DO PART 1	3	I_2	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-18 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE A, B IF A > B 1.4 TYPE B, A IF B > A 1.5 TYPE A IF A = B DO PART 1	2	I_3	E_1	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE A, B IF A > B 1.4 TYPE B, A IF B > A 1.5 TYPE A, B IF A = B DO PART 1	1	I_4	E_4	A_3	F_2
	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE "THE ORDER IS" 1.4 TYPE A, B IF B < A 1.5 TYPE B, A IF A < B 1.6 TYPE A IF A = B DO PART 1	1	I_5	E_3	A_2	F_1
	1.1 DEMAND A 1.2 TYPE A, B IF A > B 1.3 TYPE B, A IF A < B 1.4 TYPE A IF A = B DO PART 1 FOR B = ...	1	I_6	E_5	A_1	F_1
	1.1 TYPE A, B IF A > B 1.2 TYPE B, A IF B > A 1.3 TYPE A IF A = B SET A = 5 SET B = 8 DO PART 1	1	I_7	E_6	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-18 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE A, B IF A > B 1.4 TYPE B, A IF A < B 1.5 TYPE A IF A = B DO PART 1	1	I_8	E_1	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE "LARGEST AND SMALLEST" 1.4 TYPE A, B IF A > B 1.5 TYPE B, A IF A < B 1.6 TYPE A, IF A = B DO PART 1	1	I_9	E_3	A_2	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 TYPE B, A IF A < B 1.4 TYPE A, B IF B < A 1.5 TYPE A IF A = B DO PART 1	1	I_{10}	E_1	A_1	F_1
		N=18	10 classes	6 classes	3 classes	2 classes

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-21	1.1 DEMAND X 1.2 DEMAND Y 1.3 DEMAND Z 1.4 SET S = 0 1.5 SET S = 1 IF X > 0 AND Y > 0 AND Z > 0 1.6 SET S = 1 IF X < 0 AND Y < 0 AND Z < 0 1.7 SET S = 1 IF X = 0 AND Y = 0 AND Z = 0 1.8 TYPE "SAME SIGN" IF S = 1 1.9 TYPE "DIFFERENT SIGN" IF S = 0 DO PART 1, ... TIMES	17	I_1	E_1	A_1	F_1
	1.1 DEMAND X 1.2 DEMAND Y 1.3 DEMAND Z 1.4 SET S = 0 1.5 SET S = 1 IF X > 0 AND Y > 0 AND Z > 0 1.6 SET S = 1 IF X < 0 AND Y < 0 AND Z < 0 1.7 SET S = 1 IF X = 0 AND Y = 0 AND Z = 0 1.8 TYPE "SAME SIGN" IF S = 1 1.9 TYPE "DIFFERENT SIGN" IF S = 0 DO PART 1	4	I_2	E_2	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by....			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-21 (cont.)	1.1 DEMAND X 1.2 DEMAND Y 1.3 DEMAND Z 1.4 SET S = 0 1.5 SET S = 1 IF X = 0 AND Y = 0 AND Z = 0 1.6 SET S = 1 IF X > 0 AND Y > 0 AND Z > 0 1.7 SET S = 1 IF X < 0 AND Y < 0 AND Z < 0 1.8 TYPE "SAME SIGN" IF S = 1 1.9 TYPE "DIFFERENT SIGN" IF S = 0 DO PART 1	1	I_3	E_2	A_1	F_1
	1.1 DEMAND X 1.2 DEMAND Y 1.3 DEMAND Z 1.4 SET S = 0 1.5 SET S = 1 IF X < 0 AND Y < 0 AND Z < 0 1.6 SET S = 1 IF X > 0 AND Y > 0 AND Z > 0 1.7 SET S = 1 IF X = 0 AND Y = 0 AND Z = 0 1.8 TYPE "SAME SIGN" IF S = 1 1.9 TYPE "DIFFERENT SIGN" IF S = 0 DO PART 1	1	I_4	E_2	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L15-21 (cont.)	1.1 DEMAND X 1.2 DEMAND Y 1.3 DEMAND Z 1.4 SET S = 0 1.5 SET S = 1 IF X < 0 AND Y < 0 AND Z < 0 1.6 SET S = 1 IF X > 0 AND Y > 0 AND Z > 0 1.7 SET S = 1 IF X = 0 AND Y = 0 AND Z = 0 1.8 TYPE "SAME SIGN" IF S = 1 1.9 TYPE "DIFFERENT SIGN" IF S = 0 DC PART 1, ... TIMES	1	I_5	E_1	A_1	F_1
	1.1 DEMAND X 1.2 DEMAND Y 1.3 SET S = 0 1.4 SET S = 1 IF X > 0 AND Y > 0 AND Z > 0 1.5 SET S = 1 IF X < 0 AND Y < 0 AND Z < 0 1.6 SET S = 1 IF X = 0 AND Y = 0 AND Z = 0 1.7 TYPE "SAME SIGN" IF S = 1 1.8 TYPE "DIFFERENT SIGN" IF S = 0 DC PART 1 FOR Z = ...	1	I_6	E_3	A_1	F_1
		N=25	6 classes	3 classes	1 class	1 class

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-4	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 TYPE 3.1416*R↑2 2.1 TYPE "ERROR" DO PART 1	11	I_1	E_1	A_1	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 TYPE 3.1416*R↑2 2.1 TYPE "ERROR" DO PART 1, ... TIMES	7	I_2	E_2	A_1	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 TYPE R↑2*3.1416 2.1 TYPE "ERROR" DO PART 1, ... TIMES	2	I_3	E_2	A_1	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 TYPE R↑2*3.1416 2.1 TYPE "ERROR" DO PART 1	1	I_4	E_1	A_1	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 TYPE R↑2*(22/7) 2.1 TYPE "ERROR" DO PART 1	1	I_5	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-4 (cont.)	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 SET A = R↑2*3.1416 1.4 TYPE A 2.1 TYPE "ERROR" DO PART 1	1	I_6	E_3	A_2	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 LET A = R↑2*3.1416 1.4 TYPE A 2.1 TYPE "ERROR" DO PART 1	1	I_7	E_5	A_1	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 SET A = R↑2*3.1416 1.4 TYPE A 2.1 TYPE "ERROR" DO PART 1, ... TIMES	1	I_8	E_4	A_2	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 SET A = 3.1416*R↑2 1.4 TYPE A 2.1 TYPE "ERROR" DO PART 1	1	I_9	E_3	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-4 (cont.)	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 SET P = 3.1416 1.4 SET A = P*(R↑2) 1.5 TYPE A 2.1 TYPE "ERROR" 2.2 TO PART 1 DO PART 1, ... TIMES	1	I_{10}	E_4	A_3	F_1
	1.1 DEMAND R 1.2 TO PART 2 IF R < 0 1.3 TYPE "RADIUS EQUALS" 1.4 TYPE R 1.5 TYPE "AREA EQUALS" 1.6 TYPE 3.1416*R↑2 2.1 TYPE "ERROR" DO PART 1, ... TIMES	1	I_{11}	E_6	A_4	F_2
	1.1 DEMAND R AS "RADIUS" 1.2 TO PART 2 IF R < 0 1.3 SET A = (R↑2)*3.1416 1.4 TYPE A 1.5 TO STEP 1.1 2.1 TYPE "ERROR" 2.2 TO PART 1 DO PART 1	1	I_{12}	E_7	A_2	F_1
		N=29	12 classes	7 classes	4 classes	2 classes

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-6	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 TO PART 2 IF A < B AND A < C 1.5 TO PART 3 IF B < A AND B < C 1.6 TYPE C, A, B IF A <= B 1.7 TYPE C, B, A IF B < A 2.1 TYPE A, B, C IF B <= C 2.2 TYPE A, C, B IF C < B 3.1 TYPE B, A, C IF A <= C 3.2 TYPE B, C, A IF C < A DO PART 1	1	I_1	E_1	A_1	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 TO PART 2 IF A < B 1.4 TO PART 3 IF B <= A 2.1 DEMAND C 2.2 TO PART 4 IF C < A 2.3 TYPE A, B, C IF B < C 2.4 TYPE A, C, B IF C <= B 3.1 DEMAND C 3.2 TO PART 4 IF C < B 3.3 TYPE B, C, A IF C < A 3.4 TYPE B, A, C IF A <= C 4.1 TYPE C, A, B IF A < B 4.2 TYPE C, B, A IF B <= A DO PART 1	1	I_2	E_2	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-6 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 SET L = A IF A > B AND A > C 1.41 SET L = B IF B > A AND B > C 1.42 SET L = C IF C > A AND C > B 1.5 SET S = A IF A < B AND A < C 1.51 SET S = B IF B < A AND B < C 1.52 SET S = C IF C < A AND C < B 1.6 SET M = A IF A < B AND A < C 1.61 SET M = A IF A < C AND A > B 1.62 SET M = B IF B < A AND B > C 1.63 SET M = B IF B < C AND B > A 1.64 SET M = C IF C < A AND C > B 1.65 SET M = C IF C < B AND C > A 1.7 TYPE "THE NUMBERS ARE LISTED FROM" 1.71 TYPE "LARGEST TO SMALLEST" 1.72 TYPE L, M, S DO PART 1, ... TIMES	1	I_3	E_3	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...: Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-6 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 TO PART 2 2.1 TYPE A IF A < B AND A < C 2.2 TYPE B IF B < A AND B < C 2.3 TYPE C IF C < A AND C < B 2.4 TO PART 3 3.1 TYPE A IF A < C AND A > B CR A > C AND A < B 3.2 TYPE B IF B < C AND B > A OR B > C AND B < A 3.3 TYPE C IF C < B AND C > A OR C > B AND C < A 3.4 TO PART 4 4.1 TYPE A IF A > B AND A > C 4.2 TYPE B IF B > A AND B > C 4.3 TYPE C IF C > A AND C > B DO PART 1, ... TIMES	1	I_4	E_4	A_1	F_1
	10.1 DEMAND A 10.2 DEMAND B 10.3 DEMAND C 10.4 DO PART 20 IF A < B < C 10.5 DO PART 30 IF A < C < B 10.6 DO PART 40 IF B < A < C 10.7 DO PART 50 IF B < C < A 10.8 DO PART 60 IF C < A < B 10.9 DO PART 70 IF C < B < A 20.1 TYPE A, B, C 30.1 TYPE A, C, B 40.1 TYPE B, A, C 50.1 TYPE B, C, A 60.1 TYPE C, A, B 70.1 TYPE C, B, A DO PART 10, ... TIMES	1	I_5	E_5	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-6 (cont.)	1.1 DEMAND A	1	I_6	E_6	A_3	F_2
	1.2 DEMAND B					
	1.3 DEMAND C					
	1.4 TO PART 2 IF A > B AND B > C					
	1.5 TO PART 3 IF A > C AND C > B					
	1.6 TO PART 4 IF B > A AND A > C					
	1.7 TO PART 5 IF B > C AND C > A					
	1.8 TO PART 6 IF C > A AND A > B					
	1.9 TO PART 7 IF C > B AND B > A					
	2.1 TYPE A					
	2.2 TYPE B					
	2.3 TYPE C					
	3.1 TYPE A					
	3.2 TYPE C					
	3.3 TYPE B					
	4.1 TYPE B					
	4.2 TYPE A					
	4.3 TYPE C					
	5.1 TYPE B					
	5.2 TYPE C					
	5.3 TYPE A					
	6.1 TYPE C					
	6.2 TYPE A					
	6.3 TYPE B					
	7.1 TYPE C					
	7.2 TYPE B					
	7.3 TYPE A					
	DO PART 1, ... TIMES					

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L16-6 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 SET S = A 1.5 SET S = B IF B < S 1.6 SET S = C IF C < S 1.7 TO PART 2 IF S = A 1.8 TO PART 3 IF S = B 1.9 TO PART 4 IF S = C 2.1 TYPE A, B, C IF B < C 2.2 TYPE A, C, B IF C <= B 3.1 TYPE B, A, C IF A < C 3.2 TYPE B, C, A IF C <= A 4.1 TYPE C, A, B IF A < B 4.2 TYPE C, B, A IF B <= A DO PART 1, ... TIMES	1	I_7	E_7	A_4	F_1
	9.1 DEMAND A 9.2 DEMAND B 9.3 DEMAND C 9.4 TO PART 10 10.1 TO PART 12 IF A > B AND A > C 10.2 TO PART 13 IF B > A AND B > C 10.3 TO PART 14 IF C > A AND C > B 12.1 TYPE C, B, A IF B > C 12.2 TYPE B, C, A IF B <= C 13.1 TYPE C, A, B IF A > C 13.2 TYPE A, C, B IF A <= C 14.1 TYPE B, A, C IF A > B 14.2 TYPE A, B, C IF A <= B DO PART 9	1	I_8	E_8	A_5	F_1
		N=8	8 classes	8 classes	5 classes	2 classes

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L23-7	1.1 SET X = 1 1.2 TYPE X, 2*X, 3*X, 4*X, 5*X IN FORM 2 1.3 SET X = X + 1 1.4 TO STEP 1.2 IF X <= 5 FORM 2: ←← ←← ←← ←← ←← DO PART 1	2	I_1	E_1	A_2	F_1
	1.1 TYPE "MULTIPLICATION TABLE" 1.2 TYPE ← 1.3 SET X = 1 1.4 TYPE X, 2*X, 3*X, 4*X, 5*X IN FORM 2 1.5 SET X = X + 1 1.6 TO STEP 1.4 IF X < 6 FORM 2: ←← ←← ←← ←← ←← DO PART 1	2	I_2	E_2	A_1	F_1
	1.1 TYPE "MULTIPLICATION TABLE" 1.2 SET X = 1 1.3 TYPE X, 2*X, 3*X, 4*X, 5*X IN FORM 2 1.4 SET X = X + 1 1.5 TO STEP 1.3 IF X < 6 FORM 2: ←← ←← ←← ←← ←← DO PART 1	1	I_3	E_3	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence

L23-7 (cont.)	`1.1 TYPE "MULTIPLICATION TABLE"`
	`1.2 TYPE ←`
	`1.3 TYPE ←`
	`1.4 TYPE ←`
	`1.5 TYPE ←`
	`1.6 SET X = 1`
	`1.7 TYPE X, 2*X, 3*X, 4*X, 5*X`
	`    IN FORM 2`
	`1.8 SET X = X + 1`
	`1.9 TO STEP 1.7 IF X < 6`
	`FORM 2:`
	`←← ←← ←← ←← ←←`
	`DO PART 1`
		1	I_4	E_4	A_1	F_1

	`1.1 TYPE "MULTIPLICATION TABLE"`
	`1.2 SET X = 1`
	`1.3 TYPE X, 2*X, 3*X, 4*X, 5*X`
	`    IN FORM 2`
	`1.4 SET X = X + 1`
	`1.5 TO STEP 1.3 IF X < 6`
	`1.6 TYPE "THE END"`
	`FORM 2:`
	`←← ←← ←← ←← ←←`
	`DO PART 1`
		1	I_5	E_5	A_3	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L23-7 (cont.)	1.1 DEMAND X 1.2 TYPE X*1, X*2, X*3, x*4, X*5 IN FORM 2 1.3 SET X = X + 1 1.4 TO STEP 1.2 IF X < 6 1.5 TYPE "THE END" FORM 2: ←← ←← ←← ←← ←← DO PART 1	1	I_6	E_6	A_4	F_2
	1.1 SET Y = 1 1.2 SET X = 1 1.3 TYPE X*Y 1.4 SET X = X + 1 1.5 TO STEP 1.3 IF X < 6 1.6 SET Y = Y + 1 1.7 TO STEP 1.2 IF Y < 6 DO PART 1	1	I_7	E_7	A_5	F_1
	1.1 SET X = 1 1.2 SET Y = 1 1.3 TYPE X*Y, X*(Y+1), X*(Y+2) X*(Y+3), X*(Y+4) 1.4 TYPE ← 1.5 SET X = X + 1 1.6 TO STEP 1.3 IF X < 6 DO PART 1	1	I_8	E_8	A_6	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L23-7 (cont.)	1.1 SET X = 1 1.2 SET Y = 1 1.25 TYPE "MULTIPLICATION TABLE" 1.3 TYPE X, Y, X*Y IN FORM 2 1.4 SET X = X + 1 1.5 TO STEP 1.7 IF X = 6 1.6 TO STEP 1.3 IF X < 6 1.7 SET Y = Y + 1 1.8 TYPE "TEXT" IF X = 10 1.9 TO STEP 1.3 IF Y < 6 FORM 2: ←← ←← ←←	1	I_9	E_9	A_7	F_3
	1.1 SET X = 1 1.2 TYPE X*1, X*2, X*3, X*4, X*5 1.3 SET X = X + 1 1.4 TO STEP 1.2 IF X <= 5 DO PART 1	1	I_{10}	E_{10}	A_2	F_1
	1.1 SET X = 0 1.2 TYPE "MULTIPLICATION TABLE" 1.3 TYPE X, 2*X, 3*X, 4*X, 5*X IN FORM 2 1.4 SET X = X + 1 1.5 TO STEP 1.3 IF X ≤ 6 FORM 2: 0*X = 0, 1*X = ←, 2*X = ←←, 3*X = ←←, 4*X = ←←, 5*X = ←← DO PART 1, ... TIMES	1	I_{11}	E_{11}	A_1	F_1
		N=13	11 classes	11 classes	7 classes	3 classes

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L24-11	1.1 DEMAND N 1.2 SET C = 1 1.3 TYPE C 1.4 SET C = C + 1 1.5 TO STEP 1.3 IF C <= N DO PART 1	11	I_1	E_1	A_1	F_1
	1.1 DEMAND N 1.2 SET C = 0 1.3 SET C = C + 1 1.4 TYPE C 1.5 TO STEP 1.3 IF C < N DO PART 1	2	I_2	E_2	A_2	F_1
	1.1 DEMAND N 1.2 SET C = 1 1.3 TYPE C IN FORM 2 1.4 SET C = C + 1 1.5 TO STEP 1.3 IF C <= N FORM 2: ← ← ← ← DO PART 1	2	I_3	E_3	A_1	F_1
	1.1 DEMAND N 1.2 SET C = 1 1.3 TYPE C 1.4 SET C = C + 1 1.5 TO STEP 1.3 IF C <= N DO PART 1, ... TIMES	1	I_4	E_4	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L24-11 (cont.)	1.1 DEMAND N 1.2 SET M = 1 1.3 TYPE M IN FORM 2 1.4 SET M = M + 1 1.5 TO STEP 1.3 IF M <= N 1.6 TYPE "THE END" FORM 2: ← ← ← ← DO PART 1, ... TIMES	1	I_5	E_5	A_3	F_1
	1.1 DEMAND N 1.2 SET C = 1 1.3 TYPE C 1.4 TO PART 2 IF C = N 1.5 SET C = C + 1 1.6 TO STEP 1.3 2.1 TYPE "THE END" DO PART 1	1	I_6	E_6	A_3	F_1
		N=18	6 classes	6 classes	3 classes	1 class
L25-8	1.1 TYPE 1/C DO STEP 1.1 FOR C = 1(1)7	15	I_1	E_1	A_1	F_1
	1.1 TYPE 1/C DO PART 1 FOR C = 1,2,3,4,5,6,7	4	I_2	E_1	A_1	F_1
	1.1 TYPE 1/C DO PART 1 FOR C = 1(1)7	2	I_3	E_1	A_1	F_1
	1.1 TYPE 1/C DO STEP 1.1 FOR C = 1,2,3,4,5,6,7	2	I_4	E_1	A_1	F_1
		N=23	4 classes	1 class	1 class	1 class

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L26-5	1.1 TYPE "HOW MANY INCHES?" 1.2 DEMAND I 1.3 SET F = IP(I/12) 1.4 SET I = I - 12*F 1.5 TYPE F, I IN FORM 2 1.6 TO STEP 1.2 FORM 2: ← ← FEET, ← ← INCHES DO PART 1	6	I_1	E_1	A_1	F_1
	1.1 TYPE "HOW MANY INCHES?" 1.2 DEMAND I 1.3 SET F = IP(I/12) 1.4 SET I = I - 12*F 1.5 TYPE F, I IN FORM 2 1.6 TO STEP 1.1 FORM 2: ← ← FEET, ← ← INCHES DO PART 1	5	I_2	E_1	A_1	F_1
	1.1 DEMAND X 1.2 SET F = IP(X/12) 1.3 SET I = X - F*12 1.4 TYPE F, I IN FORM 1 1.5 TO STEP 1.1 FORM 1: ← ← FEET, ← ← INCHES DO PART 1	2	I_3	E_3	A_3	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L26-5 (cont.)	1.1 DEMAND I 1.2 SET F = IP(I/12) 1.3 SET I = I - F*12 1.4 TYPE F, I 1.5 TO STEP 1.1 DO PART 1	1	I_4	E_2	A_2	F_1
	1.1 DEMAND I 1.2 SET F = IP(I/12) 1.3 SET I = I - 12*F 1.4 TYPE F, I 1.5 TO STEP 1.1 DO PART 1	1	I_5	E_2	A_2	F_1
	1.1 DEMAND I 1.2 SET F = IP(I/12) 1.3 SET I = I - F*12 1.4 TYPE F 1.5 TYPE I 1.6 TO STEP 1.1 DO PART 1	1	I_6	E_2	A_2	F_1
	1.1 DEMAND I 1.2 SET F = IP(I/12) 1.3 SET I = I - F*12 1.4 TYPE F, I IN FORM 2 1.5 TO STEP 1.1 FORM 2: ←← FEET, ←← INCHES DO PART 1	1	I_7	E_4	A_2	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L26-5 (cont.)	1.1 DEMAND I 1.2 SET F = IP(I/12) 1.3 SET I = I - 12*F 1.4 TYPE F, I IN FORM 2 1.5 TO STEP 1.1 FORM 2: ←←FEET, ←←INCHES DO PART 1	1	I_8	E_4	A_2	F_1
	1.1 TYPE "HOW MANY INCHES?" 1.2 DEMAND X 1.3 SET F = IP(X/12) 1.4 SET I = X - (F*12) 1.5 TYPE X, F, I IN FORM 2 1.6 TO STEP 1.2 FORM 2: ←←INCHES EQUALS ←←FEET, ←←INCHES DO PART 1	1	I_9	E_5	A_4	F_2
	1.1 DEMAND I 1.2 TYPE IP(I/12), I - 12*IP(I/12) IN FORM 2 1.3 TO STEP 1.1 FORM 2: ←←FEET, ←← . ←←← INCHES DO PART 1	1	I_{10}	E_6	A_5	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L26-5 (cont.)	1.1 DEMAND X 1.2 SET F = IP(X/12) 1.3 SET I = FP(X/12)*12 1.4 TYPE X; F, I IN FORM 2 1.5 TO STEP 1.1 FORM 2: ← ← ← INCHES EQUALS ← ← FEET AND ← ← INCHES DO PART 1	1	I_{11}	E_7	A_6	F_2
	1.1 TYPE "HOW MANY INCHES?" 1.2 DEMAND X AS "INCHES" 1.3 SET F = IP(X/12) 1.4 SET I = X - (12*F) 1.5 TYPE F, I IN FORM 2 1.6 TO STEP 1.1 FORM 2: ← ← FEET, ← ← INCHES DO PART 1	1	I_{12}	E_8	A_3	F_1
		N=22	12 classes	8 classes	6 classes	2 classes
L29-19	1.1 SET R = 13/7 1.2 DEMAND A 1.3 DEMAND B 1.4 DEMAND C 1.5 TYPE "A 1.5 TYPE "A ICT 13/17" IF !A-R! < !B-R! AND !A-R! < !B-R! 1.6 TYPE "B ICT 13/17" IF !B-R! < !A-R! AND !B-R! < !C-R! 1.7 TYPE "C ICT 13/17" IF !C-R! < !A-R! AND !C-R! < !B-R! DO PART 1	2	I_1	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L29-19 (cont.)	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 SET K = 13/17 1.5 SET X = !A-K! 1.6 SET Y = !B-K! 1.7 SET Z = !C-K! 1.8 TYPE FORM 1 IF X < Y AND X < Z 1.9 TYPE FORM 2 IF Y < X AND Y < Z 1.91 TYPE FORM 3 IF Z < X AND Z < Y 1.92 TO STEP 1.1 FORM 1: A IS CLOSEST TO 13/17 FORM 2: B IS CLOSEST TO 13/17 FORM 3: C IS CLOSEST TO 13/17	1	I_2	E_2	A_2	F_1
	1.1 SET R = 13/17 1.2 DEMAND A 1.3 DEMAND B 1.4 DEMAND C 1.5 TYPE FORM 1 IF !A-R! < !B-R! AND !A-R! < !C-R! 1.6 TYPE FORM 2 IF !B-R! < !A-R! AND !B-R! < !C-R! 1.7 TYPE FORM 3 IF !C-R! < !A-R! AND !C-R! < !B-R! FORM 1: A IS CLOSEST TO 13/17 FORM 2: B IS CLOSEST TO 13/17 FORM 3: C IS CLOSEST TO 13/17 DO PART 1, ... TIMES	1	I_3	E_3	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L29-19 (cont.)	1.1 TO STEP 1.5 IF !A-13/17! < !B-13/17! 1.2 TYPE FORM 1 IF !B-13/17! < !C-13/17! 1.3 TYPE FORM 2 IF !B-13/17! >= !C-13/17! 1.4 TO STEP 1.7 1.5 TYPE FORM 3 IF !A-13/17! <= !C-13/17! 1.6 TYPE FORM 2 IF !A-13/17! > !C-13/17! 1.7 TYPE "THE END" FORM 1: B IS CLOSEST TO 13/17 FORM 2: C IS CLOSEST TO 13/17 FORM 3: A IS CLOSEST TO 13/17 SET A = ... SET B = ... SET C = ... DO PART 1	1	I_4	E_4	A_3	F_1
	1.1 DEMAND A 1.2 DEMAND B 1.3 DEMAND C 1.4 SET A = !A-13/17! 1.5 SET B = !B-13/17! 1.6 SET C = !C-13/17! 1.7 TYPE "A IS CLOSEST TO 13/17" IF A < B AND A < C 1.8 TYPE "B IS CLOSEST TO 13/17" IF B < A AND B < C 1.9 TYPE "C IS CLOSEST TO 13/17" IF C < A AND C < B DO PART 1	1	I_5	E_5	A_4	F_1
		N=6	5 classes	5 classes	4 classes	1 class

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L32-5	1.1 SET S = 0 1.2 DO PART 2 FOR I = 1(1)6 1.3 TYPE S 2.1 TYPE L(I) 2.2 SET S = S + L(I) DO PART 1	2	I_1	E_1	A_1	F_1
	1.1 SET I = 1 1.2 SET S = 0 1.3 SET S = S + L(I) 1.4 TYPE L(I) 1.5 SET I = I + 1 1.6 TO STEP 1.3 IF I < 7 1.7 TYPE S DO PART 1	1	I_2	E_2	A_2	F_1'
	3.1 SET I = 1 3.2 SET S = 0 3.3 DEMAND X 3.4 SET L(I) = X 3.5 TYPE L(I) 3.6 SET I = I + 1 3.7 SET S = L(I) + S 3.8 TYPE S IF I = 7 3.9 TO STEP 3.3 IF I <= 6 DO PART 3	1	I_3	E_3	A_3	F_2

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L-32-5 (cont.)	1.1 DEMAND N 1.2 SET I = 1 1.3 DEMAND X 1.4 SET A(I) = X 1.5 SET I = I + 1 1.6 TO STEP 1.3 IF I <= N 1.7 SET I = 1 1.8 SET S = 0 1.9 TYPE A(I) 1.91 SET S = S + A(I) 1.92 SET I = I + 1 1.93 TO STEP 1.9 IF I <= N 1.94 TYPE S IN FORM 2 FORM 2: THE SUM IS ← ← ← . ← ← DO PART 1, ... TIMES	1	I_4	E_4	A_4	F_3
	1.1 SET I = 0 1.2 SET S = 0 1.3 DEMAND X 1.4 SET I = I + 1 1.5 SET S = S + X 1.6 TYPE X, S 1.7 TO STEP 1.3 IF I < 7 DO PART 1	1	I_5	E_5	A_5	F_4
	1.1 SET I = 1 1.2 SET S = 0 1.3 TYPE L(I) 1.4 SET S = S + L(I) 1.5 SET I = I + 1 1.6 TO STEP 1.3 IF I < 7 1.7 TYPE S DO PART 1	1	I_6	E_2	A_6	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L32-5 (cont.)	7.1 TYPE "THE NUMBERS TO BE ADDED" 7.2 TYPE L(1), L(2), L(3), L(4), L(5), L(6) 7.3 TYPE "THE SUM IS" 7.4 TYPE L(1) + L(2) + L(3) + L(4) + L(5) + L(6) DO PART 7	1	I_7	E_6	A_7	F_1
	2.1 TYPE L(X) DO PART 2 FOR X = 1,2,3,4,5,6 1.1 SET S = 0 1.2 TYPE S 1.3 SET I = 1 1.4 TYPE S + L(I) 1.5 SET S = S + L(I) 1.6 SET I = I + 1 1.7 TO STEP 1.4 IF I < 7 DO PART 1	1	I_8	E_7	A_8	F_5
		N=9	8 classes	7 classes	8 classes	5 classes
L32-8	4.1 SET S = 0 4.2 DO STEP 5.1 FOR I = 1(1)10 4.3 SET A = S/10 4.4 TYPE A 5.1 SET S = S + L(I) DO PART 4	7	I_1	E_1	A_1	F_1

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L32-8 (cont.)	1.1 DO PART 2 FOR I = 1(1)10 1.2 SET S = 0 1.3 DO STEP 3.1 FOR I = 1(1)10 1.4 SET A = S/10 1.5 TYPE A 2.1 DEMAND N 2.2 SET L(I) = N 3.1 SET S = S + L(I) DO PART 1, ... TIMES	2	I_2	E_2	A_2	F_2
	1.1 SET S = 0 1.2 DO STEP 2.1 FOR I = 1,2,3,4,5,6, 7,8,9,10 1.3 SET A = S/10 1.4 TYPE A 2.1 SET S = S + L(I) DO PART 1	1	I_3	E_1	A_1	F_1
	1.01 DEMAND L(1) 1.02 DEMAND L(2) 1.03 DEMAND L(3) 1.04 DEMAND L(4) 1.05 DEMAND L(5) 1.06 DEMAND L(6) 1.07 DEMAND L(7) 1.08 DEMAND L(8) 1.09 DEMAND L(9) 1.10 DEMAND L(10) 1.11 SET S = 0 1.12 DO PART 2 FOR I = 1(1)10 1.13 SET A = S/10 1.14 TYPE A 2.1 SET S = S + L(I) DO PART 1	1	I_4	E_3	A_3	F_2

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L32-8 (cont.)	1.01 DEMAND L(1) 1.02 DEMAND L(2) 1.03 DEMAND L(3) 1.04 DEMAND L(4) 1.05 DEMAND L(5) 1.06 DEMAND L(6) 1.07 DEMAND L(7) 1.08 DEMAND L(8) 1.09 DEMAND L(9) 1.10 DEMAND L(10) 1.11 SET S = 0 1.12 DO PART 2 FOR I = 1(1)10 1.13 SET A = S/10 1.14 TYPE A, S 2.1 SET S = S + L(I) DO PART 1	1	I_5	E_4	A_6	F_3
	1.1 SET S = 0 1.2 DO PART 6 FOR I = 1(1)10 1.3 DO PART 7 FOR I = 1(1)10 1.4 TYPE S/10 1.5 TO STEP 1.1 6.1 DEMAND L(I) 7.1 SET S = S + L(I) DO PART 1	1	I_6	E_5	A_5	F_2
	1.1 SET S = 0 1.2 DO PART 2 FOR I = 1(1)10 1.3 SET A = S/10 1.4 TYPE A 2.1 DEMAND X 2.2 SET L(I) = X 2.3 SET S = S + L(I) DO PART 1, ... TIMES	1	I_7	E_6	A_4	F_2

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L32-8 (cont.)	1.1 SET S = 0 1.2 DO PART 2 FOR I = 1(1)10 1.3 SET A = S/10 1.4 TYPE A 2.1 DEMAND X 2.2 SET L(I) = X 2.3 SET S = S + L(I) DO PART 1	1	I_8	E_7	A_4	F_2
	1.1 TO STEP 1.3 1.2 DEMAND A(I) 1.3 DO STEP 1.2 FOR I = 1(1)10 1.4 SET S = 0 1.5 DO STEP 2.1 FOR B = 1(1)10 1.6 SET A = S/10 1.7 TYPE A 2.1 SET S = S + A(B) DO PART 1	1	I_9	E_8	A_3	F_2
		N=16	9 classes	8 classes	6 classes	3 classes
L32-19	1.1 TYPE L(I) IF L(I) < 30 DO STEP 1.1 FOR I = 1(1)10	2	I_1	E_1	A_1	F_1
	1.1 DO PART 2 FOR I = 1(1)10 1.2 TYPE L 1.3 DO PART 3 FOR I = 1(1)10 2.1 DEMAND N 2.2 SET L(I) = N 3.1 TYPE L(I) IF L(I) < 30 5.1 TO STEP 1.2 DO STEP 1.1 DO STEP 5.1	1	I_2	E_2	A_2	F_2

Problem Number	Form of Program	N	Equivalence Class when Partitioned by ...			
			Formal Identity	Formal Equivalence	Algorithmic Equivalence	Functional Equivalence
L32-19 (cont.)	1.1 TYPE L(I) IF L(I) < 30 DO PART 1 FOR I = 1(1)10	1	I_3	E_1	A_1	F_1
	1.01 SET L(1) = ... 1.02 SET L(2) = ... 1.03 SET L(3) = ... 1.04 SET L(4) = ... 1.05 SET L(5) = ... 1.06 SET L(6) = ... 1.07 SET L(7) = ... 1.08 SET L(8) = ... 1.09 SET L(9) = ... 1.10 SET L(10) = ... 1.2 DO PART 3 FOR I = 1(1)10 3.1 TYPE L(I) IF L(I) < 30 DO PART 1	1	I_4	E_3	A_3	F_3
	1.1 DO PART 8 FOR I = 1(1)10 1.2 DO PART 9 FOR K = 1(1)10 8.1 DEMAND N 8.2 SET M(I) = N 9.1 TO STEP 9.3 IF M(K) >= 30 9.2 TYPE M(K) 9.3 TYPE ← DO PART 1	1	I_5	E_4	A_4	F_2
		N=6	5 classes	4 classes	4 classes	3 classes

APPENDIX C*

The Theory of Diversity and Coherence

*This appendix is an excerpt from J. E. Friend & M. T. Kane, "Diversity of Coherence," (in preparation).

Definitions 1 to 5 and Theorems 1 to 7 are standard mathematical notions and are included for completeness and clarity. Definitions 6 and 7 and Theorems 8 to 17 are also found in the mathematical literature, but are less well known. In this exposition, all sets are taken to be countable and all random variables discrete. In many instances definitions and proofs are given only for finite sets, although it is clear that most of the theorems hold for at least countable sets.

It is assumed that the reader is familiar with the elements of statistics, and that he has some acquaintance with set notation and the basic properties of inclusion, union, intersection, and set difference. Some of the standard theorems that are assumed in the following are:

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

$$A \cup (B \cup C) = (A \cup B) \cup C$$

$$A \subseteq A \cup B$$

$$A \cap B \subseteq A \cup B$$

If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$

If $A \subseteq B$ then $A \cap B = A$ and $A \cup B = B$.

For the cardinality of a set X, the notation N(X) is used, and these theorems (among others) are assumed:

If $A \subseteq B$ then $N(A) \leq N(B)$

If $A \cap B = \phi$ then $N(A) + N(B) = N(A \cup B)$

$$N(A) + N(B) = N(A \cup B) - N(A \cap B)$$

$$N(\phi) = 0 \, .$$

The casual reader may wish to omit proofs. A reader who is familiar with simple properties of equivalence relations may start reading with the comment preceding Definition 6. The comments are not essential and may be omitted by any reader.

Definition 1. A classification of a set S is a sequence $S_1, S_2, \ldots, S_k$ with these properties:

(i) $S_i \subseteq S$ for each i

(ii) $S_1 \cup S_2 \cup \ldots \cup S_k = S$

(iii) $S_i \cap S_j = \phi$ if $i \neq j$.

Comment. A classification is a means of subdividing a set S into subsets so that each member of S is in one and only one of the subsets. (This definition of classification is similar, but not identical, to the more commonly used partition. In a partition, empty sets are not allowed, whereas one or more of the sets in a classification may be empty.)

Theorem 1. Suppose $S_1, S_2, \ldots, S_k$ is a classification of S and that $S' \subseteq S$. Then $S_1 \cap S', S_2 \cap S', \ldots, S_k \cap S'$ is a classification of S'.

Proof:

Proof of (i): $S_i \cap S' \subseteq S'$ since $A \cap B \subset B$ for any sets A and B.

Proof of (ii): $(S_1 \cap S') \cup (S_2 \cap S') \cup \ldots (S_k \cap S') = (S_1 \cup S_2 \cup \ldots \cup S_k) \cap S' = S \cap S'$. Now $S \cap S' = S'$ since $S' \subseteq S$, so (ii) is established.

Proof of (iii): We write $S_i \cap S'$ as S_i' for each $i \leq k$ so that $S_1 \cap S', S_2 \cap S', \ldots, S_k \cap S' = S_1', S_2', \ldots, S_k'$. Thus we have $S_i' \cap S_j' = (S_i \cap S') \cap (S_j \cap S') = (S_i \cap S_j) \cap S'$. If $i \neq j$ we have $S_i \cap S_j = \phi$ so $(S_i \cap S_j) \cap S' = \phi$. Hence, $S_i' \cap S_j' = \phi$.

Theorem 2. Suppose $S_1, S_2, \ldots, S_k$ is a classification of a set S. Then $S_1, S_2, \ldots, S_{k-1}$ is a classification of $S - S_k$.

Proof: For each $i < k$ we have $S_i \cap (S-S_k) = (S_i \cap S) - (S_i \cap S_k) = S_i \cap S = S_i$. For $i = k$, $S_i \cap (S-S_k) = (S_k \cap S) - (S_k \cap S_k) = S_k - S_k = \phi$. Hence, $S_1, S_2, \ldots, S_{k-1}, \phi = S_1 \cap (S-S_k), S_2 \cap (S-S_k), \ldots, S_k \cap (S-S_k)$, which is a classification of $S-S_k$ by Theorem 1. It is easily verified from the definition of a classification that if $S_1, S_2, \ldots, S_k, \phi$ is a classification of S, so is $S_1, S_2, \ldots, S_k$, and the proof is complete.

Theorem 3. Suppose $S_1, S_2, \ldots, S_k$ is a classification of a set S. Then

$$\sum_{i=1}^{k} N(S_i) = N(S) ,$$

where N(S) is used to denote the number of elements in the set S.

Proof: The proof is by induction on c. If $c = 1$, then the classification contains only one set S_1. By (ii) or Definition 1 $S_1 = S$. Hence,

$$\sum_{i=1}^{k} N(S_i) = N(S_1) = N(S) .$$

Now, assume the theorem is true for $n < k$. $N(S) = N(S_1 \cup S_2 \cup \ldots \cup S_k)$ by (ii) and

$$N(S) = N(S_1 \cup S_2 \cup \ldots \cup S_{k-1}) + N(S_k) -$$

$$- N((S_1 \cup S_2 \cup \ldots \cup S_{k-1}) \cap S_k)$$

since

$$N(A \cup B) = N(A) + N(B) - N(A \cap B)$$

for any sets A and B. The set $(S_1 \cup S_2 \cup \ldots \cup S_{k-1}) \cap S_k$ is empty, however, since any element in S_k cannot be in S_i for $i \neq k$ by (iii). Hence, we have

$$N(S) = N(S_1 \cup S_2 \cup \ldots \cup S_{k-1}) + N(S_k) .$$

The sequence $S_1, \ldots, S_{k-1}$ is a classification of the set $S - S_k$ from Theorem 2, so

$$N(S-S_k) = \sum_{i=1}^{k-1} N(S_i)$$

and

$$N(S) = N(S-S_k) + N(S_k) = \sum_{i=1}^{k-1} N(S_i) + N(S_k) = \sum_{i=1}^{k} N(S_i) .$$

Definition 2. An equivalence relation R on a set S is a set of ordered pairs $\{(x,y)\}$ with the following properties:

(i) If $(x,y) \in R$ then $x \in S$ and $y \in S$.

(ii) If $x \in S$ then $(x,x) \in R$.

(iii) If $(x,y) \in R$ then $(y,x) \in R$.

(iv) If $(x,y) \in R$ and $(y,z) \in R$ then $(x,z) \in R$.

Comment. An equivalence relation contains the pairs (x,y) for which x is equivalent to y. Thus (i) states that we are concerned only with pairs from the given set S; (ii), which is known as the reflexive property, states that every element in S is equivalent to itself; (iii) says that if x is equivalent to y, then y is equivalent to x (this is sometimes phrased 'an equivalence relation is symmetric'. (iv) states that if x is equivalent to y and y is equivalent to z, then x is equivalent to z (this is the transitive property, which is shared by a large number of relations that are not equivalence relations ($<$, $\geq$, $\subset$, etc.)).

Theorem 4. Suppose $S_1, S_2, \ldots, S_k$ is a classification of S. Then $R = \{(x,y): x,y \in S_i$ for some i such that $1 \leq i \leq k\}$ is an equivalence relation.

Proof:

Proof of (i): If $(x,y) \in R$ then $x \in S_i$ and $y \in S_i$ for some i. Hence, $x \in S_1 \cup S_2 \cup \ldots \cup S_k$ and $y \in S_1 \cup S_2 \cup \ldots \cup S_k$. By Definition 1, $S_1 \cup S_2 \cup \ldots \cup S_k = S$. Hence, $x \in S$ and $y \in S$.

Proof of (ii): If $x \in S$ then $x \in S_i$ for some i by (ii) of Definition 1. Hence, $(x,x) \in R$.

Proof of (iii): If $(x,y) \in R$ then $x \in S_i$ and $y \in S_i$ for some i. Hence, $y \in S_i$ and $x \in S_i$ so $(y,x) \in R$.

Proof of (iv): If $(x,y) \in R$ and $(y,z) \in R$ then $x \in S_i$ and $y \in S_i$ for some i and $y \in S_j$ and $z \in S_j$ for some j. Hence, $y \in S_i \cap S_j$ so $S_i \cap S_j$ is not empty. By (iii) of Definition 1, $i = j$. Hence, $x \in S_i$ and $z \in S_j = S_i$ so $(x,z) \in R$.

Comment. From Theorem 4 we know that every classification 'defines' a unique equivalence relation that is formed by taking all possible pairs from S_1, including those of the form (x,x), together with all possible pairs from S_2, etc. The classes $S_1, S_2, \ldots$ are known as equivalence classes. The converse of Theorem 4 is also true, although it is not proved here; every equivalence relation defines a classification. (The classification defined by an equivalence is not unique, because the order of classes may vary and because one or more empty sets are allowed. However, it is essentially unique, i.e., unique up to empty sets and order.) In the following it may be assumed that every equivalence relation is defined by means of some classification.

Theorem 5. Suppose $S_1, S_2, \ldots, S_k$ is a classification of S, and $R = \{(x,y): x,y \in S_i$ for some $i\}$. Then

$$N(R) = \sum_{i=1}^{k} N^2(S_i) .$$

Proof: (deferred)

Comment. From the above theorem, we know that the number of elements (pairs) in an equivalence relation is the sum of the squares of the numbers of elements in the equivalence classes. The proof of this theorem depends upon a more general theorem concerning the number of elements in the cross product of two sets:

Definition 3. If S and S' are two sets, the cross product of $S \times S'$ is $\{(x,y): x \in S$ and $y \in S'\}$.

Lemma. $N(S \times S') = N(S) \cdot N(S')$

Proof: The proof is by induction on N(S).

If $N(S) = 1$ then $S = \{s\}$ for some s and $S \times S' = \{(s,y): y \in S'\}$. Let $f(x) = (s,x)$ for $x \in S'$. Then $f(x)$ is a one-one correspondence between S' and $S \times S'$ so $N(S') = N(S \times S')$. Since $N(S) = 1$ we have $N(S \times S') = N(S) \cdot N(S')$.

Assume $N(S \times S') = N(S) \cdot N(S')$ if $N(S) < n$, and let $S'' = \{s_1, s_2, \ldots, s_n\}$ be a set with n members. Then $S'' \times S' = \{(x,y): x \in S''$ and $y \in S'\} = \{(x,y): x \in S'' - \{s_n\}$ and $y \in S'\} \cup \{(s_n,y): y \in S'\}$, which is a union of disjoint sets since $s_n \notin S'' - \{s_n\}$. Hence,

$$\begin{aligned}
N(S'' \times S') &= N(\{(x,y): x \in S'' - \{s_n\} \text{ and } y \in S'\}) \\
&\quad + N(\{(s_n,y): y \in S'\}) \\
&= N((S'' - \{s_n\}) \times S') + N(\{s_n\} \times S') \\
&= N(S'' - \{s_n\}) \cdot N(S') + N(\{s_n\}) \cdot N(S') \\
&= [N(S'' - \{s_n\}) + N(\{s_n\})] \cdot N(S') \\
&= N(S'') \cdot N(S') .
\end{aligned}$$

<u>Proof of Theorem 5</u>. We prove that $N(R) = \sum_{i=1}^{k} N^2(S_i)$ as follows. Let $S_i \times S_i = \{(x,y): x,y \in S_i\}$ be the cross product of S_i with itself. From the definitions of R and $S_i \times S_i$ it is readily seen that $R \subseteq S_1 \times S_1 \cup S_2 \times S_2 \cup \ldots \cup S_k \times S_k$. Suppose $(x,y) \in S_1 \times S_1 \cup S_2 \times S_2 \cup \ldots \cup S_k \times S_k$. Then $(x,y) \in S_i \times S_i$ for some i. Hence, $x \in S_i$ and $y \in S_i$ so $(x,y) \in R$ by the definition of R, and we have shown that $S_1 \times S_1 \cup S_2 \times S_2 \cup \ldots \cup S_k \times S_k \subseteq R$. Since the inclusion holds both ways, the two sets are equal.

Now $S_i \times S_i \cap S_j \times S_j = \phi$ if $i \neq j$, since $(x,y) \in S_i \times S_i$ and $(x,y) \in S_j \times S_j$ imply that x and y are elements of both S_i and S_j, which is impossible by (iii) of Definition 1.

Hence, R is a union of disjoint sets, so $N(R) = \sum_{i=1}^{k} N(S_i \times S_i)$. From the lemma, $N(S_i \times S_i) = N(S_i) \cdot N(S_i)$ so

$$N(R) = \sum_{i=1}^{k} N^2(S_i) .$$

Theorem 6. If $R \subseteq R'$ then $\sum_{i=1}^{k} N^2(S_i) \leq \sum_{i=1}^{k'} N^2(S_i')$.

Proof: Since $R \subseteq R'$,

$$N(R) \leq N(R') .$$

Hence,

$$\sum_{i=1}^{k} N^2(S_i) \leq \sum_{i=1}^{k'} N^2(S_i') \quad \text{by Theorem 5.}$$

Definition 4. The identity relation I on a set S is $\{(x,x): x \in S\}$.

Theorem 7. I is an equivalence relation.

Comment. If a set S has n elements the relation I contains n pairs of the form (x,x) and the classification defined ~~by I~~ is n sets of the form $\{x\}$. Under the identity relation no element is equivalent to any element other than itself. The opposite of this is the universal relation U, defined below, under which any two elements in S are equivalent.

Definition 5. The universal relation U on a set S is $\{(x,y): x,y \in S\}$.

Theorem 8. U is an equivalence relation.

Comment. If a set S has n members then U contains n^2 pairs. The classification defined by U consists of only one subset of S, namely, S itself.

Theorem 9. For any equivalence relation R,

$$I \subseteq R \subseteq U .$$

Proof: If $(x,y) \in I$ then $y = x \in S$. Hence, $(x,y) \in R$. If $(x,y) \in R$ then $x \in S$ and $y \in S$ so $(x,y) \in U$.

Comment. In the above theorem, as in many of the following, it is assumed that all equivalence relations are defined on the same set S.

Theorem 10. If R and R' are equivalence relations, then $R \cap R'$ is an equivalence relation.

Proof (referring to Definition 2):

Proof of (i): If $(x,y) \in R \cap R'$ then $(x,y) \in R$ so x and y are in S.

Proof of (ii): If $x \in \in S$ then $(x,x) \in R$ and $(x,x) \in R'$. Hence, $(x,x) \in R \cap R'$.

Proof of (iii): If $(x,y) \in R \cap R'$ then $(x,y) \in R$ and $(x,y) \in R'$. Hence, $(y,x) \in R$ and $(y,x) \subset R'$ so $(y,x) \in R \cap R'$.

Proof of (iv): Assume $(x,y) \in R \cap R'$ and $(y,z) \in R \cap R'$. Then $(x,y) \in R$ and $(y,z) \in R$ so $(x,z) \in R$. Similarly, $(x,y) \in R'$. Hence, $(x,z) \in R \cap R'$.

Comment. Although $R \cap R'$ is an equivalence relation, it is not always the case that $R \cup R'$ is an equivalence relation. However, $R \cup R'$ does satisfy (i), (ii), and (iii) of an equivalence relation. To satisfy (iv) we add enough pairs so that $R \cup R'$ is closed under transitivity. This result we call $R \oplus R'$. In the mathematical literature the set $R \oplus R'$ is called the transitive closure of R and R'. The reader who is not interested in the technical details of this development may skip to Theorem 19.

Definition 6. Given two equivalence relations R and R', the nth extension of R and R', denoted Ext(n), is defined as follows.

(i) $\text{Ext}(1) = R \cup R'$.

(ii) $\text{Ext}(n) = \text{Ext}(n-1) \cup \{(x,y): \exists z$ such that $(x,z) \in \text{Ext}(n-1)$ and $(z,y) \in \text{Ext}(n-1)\}$.

Theorem 11. For any equivalence relations R and R', and for any n, $Ext(n) \subseteq U$.

Proof: The proof is by induction on n. For $n = 1$, $Ext(n) = R \cup R'$. Assume $x \in R \cup R'$. Then $x \in R$ or $x \in R'$. Hence, by Theorem 9, $x \in U$, and we conclude that $Ext(1) \subseteq U$.

Assume $Ext(n-1) \subseteq U$. Let (x,y) be a member of $Ext(n)$. Either $(x,y) \in Ext(n-1)$ or $(x,y) \in \{(x,y): \exists z$ such that $(x,z) \in Ext(n-1)$ and $(z,y) \in Ext(n-1)\}$. If $(x,y) \in Ext(n-1)$ then $(x,y) \in U$ by assumption. Otherwise, $(x,z) \in Ext(n-1)$ and $(z,y) \in Ext(n-1)$ for some z. By assumption then, $(x,z) \in U$ and $(z,y) \in U$. Hence, x and y are elements of S, and so $(x,y) \in U$. Q.E.D.

Theorem 12. $N(Ext(n)) \leq N(U)$ for any n.

Proof: This follows directly from Theorem 11 and the fact that, for any sets A and B, if $A \subseteq B$ then $N(A) \leq N(B)$.

Theorem 13. $N(Ext(n)) \leq N(Ext(n+1))$ for any n.

Proof: From Definition 6, we have $Ext(n-1) \subseteq Ext(n)$ for $n > 1$.

Comment. Theorem 13 shows that each extension of R and R' is at least as large as the preceding extension. Theorem 14 states that at some point the extensions do not become larger.

Theorem 14. For some n, $N(Ext(n)) = N(Ext(n+1))$.

Proof: The sequence

$$N(Ext(1)), N(Ext(2)), N(Ext(3)), \ldots$$

is a monotonically increasing sequence of integers with an upper bound of $N(U)$.

Comment. Theorem 14 states that at some point some extension of R and R' is identical in number to the next extension. A stronger result, below, is that these two extensions are identical in extent as well as number.

Theorem 15. For some n, $Ext(n) = Ext(n+1)$.

Proof: From Theorem 14, there is an n such that $N(Ext(n)) = N(Ext(n+1))$. We know $Ext(n) \subseteq Ext(n+1)$ from the definition of Ext, so there is a subset X of $Ext(n+1)$ such that $Ext(n+1) - X = Ext(n)$. Then $N(Ext(n+1)) - N(X) = N(Ext(n))$. But $N(X)$ must be 0 since $N(Ext(n)) = N(Ext(n+1))$ so $X = \phi$.

Theorem 16. For any n, $R \cup R' \subseteq Ext(n)$.

Proof: The proof is by induction on n. For $n = 1$, $Ext(n) = R \cup R'$, so $R \cup R' \subseteq Ext(n)$.

Assume $R \cup R' \subseteq Ext(n-1)$. By (ii) of Definition 6, $Ext(n-1) \subseteq Ext(n)$. Hence, by transitivity of $\subseteq$ we have $R \cup R' \subseteq Ext(n)$.

Comment. The following theorem shows that as soon as the extensions of R and R' stabilize (stop increasing in size) the result is an equivalence relation. This means we have added enough pairs to $R \cup R'$ to form an equivalence relation.

Theorem 17. If $Ext(n) = Ext(n+1)$ then $Ext(n)$ is an equivalence relation.

Proof (referring to Definition 2):

Proof of (i): If $(x,y) \in Ext(n)$ then $(x,y) \in U$ by Theorem 8. Hence, $x \in S$ and $y \in S$.

Proof of (ii): If $x \in S$ then $(x,x) \in R$ since R is an equivalence relation. Hence, $(x,x) \in R \cup R'$, and by Theorem 13, $(x,x) \in Ext(n)$.

Proof of (iii): The proof is by induction.

If $(x,y) \in Ext(1)$ then $(x,y) \in R \cup R'$ so $(x,y) \in R$ or $(x,y) \in R'$. If $(x,y) \in R$ then $(y,x) \in R$ since R is an equivalence relation. Hence, $(y,x) \in R \cup R'$ so $(y,x) \in Ext(1)$. If $(x,y) \in R'$ a similar argument holds.

Now assume that (iii) holds for m-1, and let (x,y) be a member of Ext(m). If $(x,y) \in Ext(m-1)$, then $(y,x) \in Ext(m-1)$ by assumption so $(y,x) \in Ext(m)$. If $(x,y) \notin Ext(m-1)$, then there is a z such that $(x,z) \in Ext(m-1)$ and $(z,y) \in Ext(m-1)$. Then (z,x) and (y,z) are also in Ext(m-1) by assumption so $(y,x) \in Ext(m)$.

Proof of (iv): Assume $(x,y) \in Ext(n)$ and $(y,z) \in Ext(n)$. By (ii) of Definition 6, $(x,z) \in Ext(n+1)$. But $Ext(n) = Ext(n+1)$ by hypothesis, so $(x,z) \in Ext(n)$.

<u>Comment</u>. Theorem 18 shows that once the extensions of R and R' have achieved the status of an equivalence relation, there are no further additions and all of the succeeding extensions are identical.

<u>Theorem 18</u>. <u>If</u> Ext(n) <u>is an equivalence relation then</u> $Ext(n) = Ext(m)$ <u>for</u> any $m \geq n$.

<u>Proof</u>: The proof is by induction on $k = m - n$.

If $m - n = 0$ then $Ext(m) = Ext(n)$, which is all that is needed.

Now, assume $Ext(m+k) = Ext(n)$. Then $Ext(m+k+1) = Ext(m+k) \cup \{(x,y)\colon \exists z$ such that $(x,z) \in Ext(m+k)$ and $(z,y) \in Ext(m+k)\}$. We will show that $\{(x,y)\colon \exists z$ such that $(x,z) \in Ext(m+k)$ and $(z,y) \in Ext(m+k)\} \subseteq Ext(m+k)$. Let (x,y) be a member of $\{(x,y)\colon \exists z$ such that $(x,z) \in Ext(m+k)$ and $(z,y) \in Ext(m+k)\}$. Then there is a z such that (x,z) and (z,y) are in Ext(m+k). By the inductive assumption, (x,z) and (z,y) are in Ext(n). Since Ext(n) is an equivalence relation, $(x,y) \in Ext(n)$ so $(x,y) \in Ext(m+k)$.

Comment. We have shown that the sequence of extensions of R and R' has a limit that is an equivalence relation. This limit is taken to be the sum of R and R'.

Definition 7. The sum of two equivalence relations, R and R', denoted $R \oplus R'$, is Ext(n) for some value of n such that Ext(n) is an equivalence relation.

Theorem 19. For any R and R', $R \oplus R'$ exists and is unique.

Proof: This theorem and proof are for R and R' finite. The existence of $R \oplus R'$ follows from Theorems 15 and 17, and the uniqueness follows from Theorem 18.

Theorem 20. $\oplus$ is commutative and associate.

Proof: Both of these properties follow from the analogous properties for U.

Definition 8. Two equivalence relations R and R' are independent in S if there are no $x,y \in S$ such that $x \neq y$ and $(x,y) \in R$ and $(x,y) \in R'$.

Comment. If R and R' are independent and x and y are two different members of S, then x and y will not be equivalent under both R and R'.

Theorem 21. R and R' are independent equivalence relations if and only if $R \cap R' = I$.

Proof:

Proof of the forward implication: The proof is by contradiction. Assume that R and R' are independent but $R \cap R' \neq I$. We know $I \subset R \cap R'$ from Theorem 9, so there is a pair (x,y) that is in $R \cap R'$ but not in I. Since $(x,y) \notin I$ implies that $x \neq y$, we have $(x,y) \in R$ and $(x,y) \in R'$ and $x \neq y$, which contradicts the assumption that R and R' are independent.

The proof of the reverse implication is similar.

Comment. The condition that $R \cap R' = I$ is equivalent to the definition of independence and could have been taken as the definition.

Definition 9. Two equivalence relations R and R' are interactive in S if there exists $x,y \in S$ such that $(x,y) \in R \oplus R'$ and $(x,y) \notin R$ and $(x,y) \notin R'$.

Comment. Two equivalence relations are considered to be interactive if the combination (sum) has the power to equate two elements that are not equivalent using either of the relations alone.

Theorem 22. R and R' are not interactive if and only if $R \oplus R' = R \cup R'$.

Proof:

Proof of the forward implication: If R and R' are not interactive, then there is no pair (x,y) such that $(x,y) \in R \oplus R'$ and $(x,y) \notin R \cup R'$. Hence, $R \oplus R' \subseteq R \cup R'$. We know from Theorem 16 that $R \cup R' \subseteq R \oplus R'$. Hence, $R \cup R' = R \oplus R'$.

Proof of the reverse implication: Assume R and R' are interactive. Then there is a pair (x,y) such that $(x,y) \in R \oplus R'$ and $(x,y) \notin R \cup R'$. Hence, $R \oplus R' \neq R \cup R'$.

Theorem 23. R and R' are not interactive if and only if $R \cup R'$ is an equivalence relation.

Proof: R and R' are not interactive if and only if $R \oplus R' = R \cup R'$ by Theorem 22. By Definition 6, $R \cup R' = \text{Ext}(1)$. By Definition 7 and Theorem 19, $R \oplus R' = \text{Ext}(1)$ if and only if $\text{Ext}(1)$ is an equivalence relation.

Comment. Necessary and sufficient conditions for noninteractivity are either

(1) $R \oplus R' = R \cup R'$

or

(2) $R \cup R'$ is an equivalence relation.

Either of these conditions could have been used to define noninteractivity.

We turn now to notion of the coherence of a classified set:

Definition 10. Suppose a set (population) S is classified by $S_1, S_2, \ldots, S_k$ and that the probability of an element being in equivalence class S_i is p_i. The coherence of the classified set S is defined to be

$$\gamma = \sum_{i=1}^{k} p_i^2 .$$

Comment. The value of γ depends upon a specified classification. If the equivalence relation defined by the classification is R, we sometimes use the notation γ_R. The coherence of a classified set is simply the probability that two elements drawn at random are in the same equivalence class. Corresponding to the population parameter γ there is a sample statistic c.

Definition 11. Suppose a finite set S is classified by $S_1, S_2, \ldots, S_k$ and that the corresponding equivalence relation is R. The sample coherence is

$$c = \frac{N(R)}{(N(S))^2} .$$

Comment. The formula for c can also be written

$$c = \frac{\sum_{i=1}^{k} (N(S_i))^2}{(N(S))^2} = \sum_{i=1}^{k} \left(\frac{N(S_i)}{N(S)} \right)^2 .$$

This last formulation shows more clearly the relation between c and γ, while the formulation used in the definition shows that c is the ratio of the number of equivalent pairs to the number of possible pairs. Again we use c_R to indicate the dependence of c upon the specified equivalence relation.

For research purposes, c is not a suitable estimator of γ because it is biased, and we define an estimator $\hat{c}$, which we will later show to be an unbiased estimator of γ.

<u>Definition 12</u>. <u>Suppose a set</u> S <u>is classified by</u> $S_1, S_2, \ldots, S_k$ <u>and that the corresponding equivalence relation is</u> R. <u>The</u> estimated coherence <u>of</u> S <u>is</u>

$$\hat{c} = \frac{N(R)-N(S)}{N(S)\cdot(N(S)-1)} .$$

<u>Comment</u>. $\hat{c}$ can also be written

$$\hat{c} = \frac{\sum_{i=1}^{k} (N(S_i))^2 - N(S)}{N(S)\cdot(N(S)-1)}$$

or

$$\hat{c} = \sum_{i=1}^{k} \frac{N(S_i)\cdot(N(S_i)-1)}{N(S)\cdot(N(S)-1)} .$$

From this it can be seen that $\hat{c}$ is the probability that two elements are in the same equivalence class when the drawing is <u>without</u> replacement (for c, the drawing is with replacement). The values of c and $\hat{c}$ are compared in the following theorems.

<u>Theorem 24</u>. $c_I = \frac{1}{N(S)}$ <u>and</u> $\hat{c}_I = 0$.

Proof: The value of $N(I)$, where I is the identity relation, is $N(S)$, so the theorem follows from direct substitution into the formulas given in the definitions of c and $\hat{c}$.

Theorem 25. $c_U = \hat{c}_U = 1$.

Proof: For the universal relation U, $N(U) = (N(S))^2$.

Theorem 26. $\hat{c} = \frac{N(S)}{N(S)-1}(c - \frac{1}{N(S)})$.

Proof: This can be verified by direct substitution.

Theorem 27. If R and R' are equivalence relations and $R \subseteq R'$ then $c_R \leq c_{R'}$. Further, $R = R'$ if and only if $c_R = c_{R'}$.

Proof: Since $R \subseteq R'$ we have $N(R) \leq N(R')$, and hence,

$$\frac{N(R)}{(N(S))^2} \leq \frac{N(R')}{(N(S))^2}.$$

Since R and R' are defined on the same set S, it follows that $c_R \leq c_{R'}$.

Now assume $R \subseteq R'$ and $c_R = c_{R'}$. Then $N(R) = N(R')$ so $R = R'$.

Theorem 28. If $R \subseteq R'$ then $\hat{c}_R \leq \hat{c}_{R'}$.

Proof: This follows from Theorems 26 and 27.

Theorem 29. For any equivalence relation R

$$\hat{c}_I \leq \hat{c}_R \leq \hat{c}_u$$

and

$$c_I \leq c_R \leq c_u.$$

Theorem 30. The range of $c \subseteq (0,1]$ and the range of $\hat{c} \subseteq [0,1]$.

Comment. Range is used here in the mathematical sense: the set of all values that can be assumed by the function c. The notation $(0,1]$ is used to denote the interval from 0 to 1 excluding 0 but including 1.

Similarly, [0,1] includes both 0 and 1. From Theorem 29 we see that the greatest coherence is attained when the universal relation is used. The least coherence results from using the identity relation, under which no two different elements are equivalent.

Theorem 31. $\hat{c}_{R\oplus R'} \geq \hat{c}_R$ for any equivalence relations R and R'.

Proof: Since $R \subseteq R \oplus R'$ this follows from Theorem 28.

Theorem 32. If R and R' are not interactive, then

$$c_R + c_{R'} = c_{R\oplus R'} + c_{R\cap R'} .$$

Proof: For any sets R and R'

$$N(R) + N(R') = N(R\cup R') + N(R\cap R') .$$

Since R and R' are not interactive, we have from Theorem 22 that

$$R \oplus R' = R \cup R' .$$

Hence,

$$N(R) + N(R') = N(R\oplus R') + N(R\cap R')$$

so

$$\frac{N(R)}{(N(S))^2} + \frac{N(R')}{(N(S))^2} = \frac{N(R\oplus R')}{(N(S))^2} + \frac{N(R\cap R')}{(N(S))^2} .$$

Since $R \oplus R'$ and $R \cap R'$ are equivalent relations defined on the same set as R and R', we have

$$c_R + c_{R'} = c_{R\oplus R'} + c_{R\cap R'} .$$

Theorem 33. If R and R' are not interactive,

$$\hat{c}_R + \hat{c}_{R'} = \hat{c}_{R \oplus R'} + \hat{c}_{R \cap R'} .$$

Theorem 34. If R and R' are independent and not interactive, then

$$\hat{c}_R + \hat{c}_{R'} = \hat{c}_{R \oplus R'} .$$

Proof: From Theorem 33 we have

$$\hat{c}_R + \hat{c}_{R'} = \hat{c}_{R \oplus R'} + \hat{c}_{R \cap R'}$$

since R and R' are not interactive. Since R and R' are independent, $R \cap R' = I$ by Theorem 21. By Theorem 24, $\hat{c}_I = 0$, and we have $\hat{c}_{R \cap R'} = 0$, which completes the proof.

Comment. Under the strong hypotheses that R and R' are independent and are not interactive in S, Theorem 34 shows that the sum of the coherence due to R and R' is the same as the coherence due to the sum of R and R'.

Having shown that $\hat{c}$ has several characteristics that makes it a good measure, we now show that $\hat{c}$ is also desirable as an estimator of γ. In particular, we show that $\hat{c}$ is consistent and unbiased. We assume that the sampling is with replacement.

Theorem 35.* $\hat{c}$ is an unbiased estimator of γ.

Proof: In this proof we use the following simplified notation:

*The proofs of this and all subsequent theorems were contributed by Michael Kane.

$$N = N(S)$$

$$n_i = N(S_i)$$

so that the formula for $\hat{c}$ becomes

$$\hat{c} = \frac{\sum_{i=1}^{k} n_i^2 - N}{N(N-1)} = \sum_{i=1}^{k} \frac{n_i^2 - n_i}{N(N-1)} .$$

We must show that $E(\hat{c}) = \gamma$. Observe that for $i = 1,2, \ldots, E$, the random variable n_i is binomial with parameters p_i and N, so that

$$E(n_i) = Np_i \quad \text{and} \quad \text{var}(n_i) = Np_i(1-p_i) .$$

Thus

$$E(n_i^2) = \text{var}(n_i) + E^2(n_i) = N(N-1)p_i^2 + Np_i$$

so

$$E(\hat{c}) = E\left(\sum_{i=1}^{k} \frac{n_i^2 - n_i}{N(N-1)}\right)$$

$$= \sum_{i=1}^{k} \frac{(E(n_i^2) - E(n_i))}{N(N-1)}$$

$$= \sum_{i=1}^{k} \frac{(N(N-1)p_i^2 + Np_i - Np_i)}{N(N-1)}$$

$$= \sum_{i=1}^{k} p_i^2$$

$$= \gamma .$$

Definition 13. Let $f(N)$ be a polynomial function of N. Then, $f(N) = O(N^{\alpha})$ iff the highest power of N in $f(N)$ is less than or equal to α.

Comment. The notation introduced in Definition 13 is useful in examining the asymptotic properties of functions. In the development that follows, we are interested in the limiting values of polynomials in N, as N approaches infinity. Since $f(N) = O(N^{\alpha})$, $f(N)$ does not grow faster than N^{α} as $N \to \infty$. This implies that $\lim_{N \to \infty} \frac{1}{N^{\beta}} O(N^{\alpha}) = 0$ for $\beta > \alpha$.

Lemma: Let $\{n_i\}$ be a set of random variables with a multinomial distribution. Let α and β be integer constants and let n_i and n_j (i possibly equal to j) be any two random variables from the set $\{n_i\}$. Then, for any sample size, N,

$$E(n_i^{\alpha} n_j^{\beta}) = N^{\alpha+\beta} p_i^{\alpha} p_j^{\beta} + O(N^{\alpha+\beta-1}) .$$

Proof: The proof is by induction on α and β. For $\alpha = 1$, $\beta = 0$, we have from (2) in Theorem 35,

$$\begin{aligned} E(n_i) &= Np_i \\ &= Np_i + O(N^0) \\ &= Np_i + O(1) . \end{aligned}$$

Similarly, the lemma holds for $\alpha = 0$, $\beta = 1$. For $\alpha = \beta = 1$,

$$E(n_i n_j) = \sum n_i n_j \frac{N!}{n_1!\ldots n_k!} p_1^{n_1}\ldots p_k^{n_k}$$

$$E(n_i n_j) = \sum \frac{N!}{n_1!\ldots(n_i-1)!\ldots(n_j-1)!\ldots n_k!} p_1^{n_1}\ldots p_k^{n_k}$$

$$= N(N-1)\, p_i p_j \sum \frac{(N-2)!}{n_1!\ldots(n_i-1)!\ldots(n_j-1)!\ldots n_k!}$$

$$p_1^{n_1}\ldots p_i^{n_i-1}\ldots p_j^{n_j-1}\ldots p_k^{n_k}$$

$$= N(N-1)p_i p_j$$

$$= N^2 p_i p_j - Np_i p_j ,$$

and since $p_i p_j$ is a constant for any population,

$$E(n_i n_j) = N^2 p_i p_j + O(N) .$$

Assume that the lemma is true for α and β less than θ. For $\beta < \theta$,

$$E(n_i^\theta\, n_j^\beta) = \sum n_i^\theta\, n_j^\beta \frac{N!}{n_1!\ldots n_k!} p_1^{n_1}\ldots p_k^{n_k}$$

$$= Np_i \sum n_i^{\theta-1}\, n_j^\beta \frac{(N-1)!}{n_1!\ldots(n_i-1)!\ldots n_k!}\; p_1^{n_1}\ldots p_i^{n_i-1}\ldots p_k^{n_k}$$

the summation above is simply the $E(n_i^{\theta-1}\, n_j^\beta)$ for a sample size of N-1, and θ-1 and β are both less than θ.

$$E(n_i^\theta\, n_j^\beta) = Np_i[(N-1)^{\theta-1+\beta}\; p_i^{\theta-1}\; p_j^\beta + O(N^{\theta+\beta-2})]$$

Now

$$(N-1)^{\theta-1+\beta} = N^{\theta-1+\beta} + O(N^{\theta-1+\beta-1})$$

$$= N^{\theta+\beta-1} + O(N^{\theta+\beta-2})$$

$$E(n_i^{\theta}\, n_j^{\beta}) = Np_i[N^{\theta+\beta-1}\, p_i^{\theta-1}\, p_j^{\beta} + O(N^{\theta+\beta-2})]$$

$$= N^{\theta+\beta}\, p_i^{\theta}\, p_j^{\beta} + O(N^{\theta+\beta-1}) .$$

So the lemma holds for $\alpha = \theta$, $\beta < \theta$. Similar treatments would show that the lemma also holds for $\beta = \theta$, $\alpha < \theta$ and for $\alpha = \beta = \theta$. Therefore, the lemma has been proved.

<u>Theorem 36</u>. $\hat{c}$ <u>is a consistent estimator</u>.

<u>Proof</u>: Since $\hat{c}$ is an unbiased estimator of γ, we need only show that var $(\hat{c})$ goes to zero as $N \to \infty$ in order to establish the consistency of $\hat{c}$.

$$\text{var}\ (\hat{c}) = E(\hat{c}^2) - [E(\hat{c})]^2 \qquad (1)$$

$$E(\hat{c}^2) = E\left(\frac{\sum_i n_i^2 - N}{N(N-1)}\right)^2 = \frac{1}{N^2(N-1)^2}\, E\left(\left(\sum_i n_i^2\right)^2 - 2N\sum_i n_i^2 + N^2\right)$$

$$= \frac{1}{N^2(N-1)^2}\left(E\left(\sum_i n_i^2\right)^2 - 2N\sum_i E(n_i^2) + N^2\right) \qquad (2)$$

$$= \frac{1}{N^2(N-1)^2}\left(E\left(\sum_i n_i^2\right)^2 - 2N^2(N-1)\sum_i p_i^2 - 2N^2 + N^2\right)$$

$$= \frac{1}{N^2(N-1)^2}\left(E\left(\sum_i n_i^2\right)^2 + O(N^3)\right)$$

$$E\left(\sum_i n_i^2\right)^2 = E\left(\sum_{i,j} n_i^2 n_j^2\right) = \sum_{i,j} E(n_i^2 n_j^2) .$$

Using the lemma,

$$E\left(\sum n_i^2\right)^2 = \sum_{i,j} [N^4 p_i^2 p_j^2 + O(N^3)]$$

$$= N^4 \sum_{i,j} p_i^2 p_j^2 + O(N^3)$$

$$= N^4 \left[\sum_i p_i^2\right]^2 + O(N^3) .$$

Inserting this in equation 2, we have

$$E(\hat{c}^2) = \frac{1}{N^2(N-1)^2}\left[N^4 \left[\sum p_i^2\right]^2 + O(N^3)\right] ,$$

and inserting this in equation 1, we have

$$\text{var}\ (\hat{c}) = \frac{1}{N^2(N-1)^2}\left[N^4\left(\sum p_i^2\right)^2 + O(N^3)\right] - \left(\sum p_i^2\right)^2$$

$$= \frac{1}{N^2(N-1)^2}\left(\left[N^4\left(\sum p_i^2\right)^2 + O(N^3)\right] - \left[N^4\left(\sum p_i^2\right)^2 + O(N^3)\right]\right)$$

$$= \frac{1}{N^2(N-1)^2}\ O(N^3)$$

$$\lim_{N\to\infty} \operatorname{var}(\hat{c}) = \lim_{N\to\infty}\left[\frac{1}{N^2(N-1)^2}\, O(N^3)\right] = 0$$

Comment. A concept closely related to coherence is diversity.

Definition 14. Suppose a set S is classified by $S_1, S_2, \ldots, S_k$ and that the probability that an element from S is in S_i is p_i. The diversity of the classified set S is

$$\delta = 1 - \gamma .$$

Comment. δ can also be written

$$\delta = 1 - \sum_{i=1}^{k} p_i^2 ,$$

and the related sample statistic is defined below.

Definition 15. Suppose a finite set S is classified by $S_1, S_2, \ldots, S_k$ and that the corresponding equivalence relation is R. Then the sample diversity of S is

$$d = 1 - c .$$

Definition 16. Suppose a finite set S is classified by $S_1, S_2, \ldots, S_k$ and that the corresponding equivalence relation is R. The estimated diversity of S is

$$\hat{d} = 1 - \hat{c} .$$

Comment. Most of the properties of the coherence measures γ, c, and $\hat{c}$ have simple analogies for δ, d, and $\hat{d}$. The exceptions are the 'additivity' theorems, Theorems 33 and 34. Just as $\hat{c}$ is a consistent, unbiased estimator of coherence, so is $\hat{d}$ a consistent, unbiased estimator of δ:

Theorem 37. $\hat{d}$ is an unbiased, consistent estimator of δ.

Proof:

$$\begin{aligned} \hat{d} &= 1 - \hat{c} \\ E(\hat{d}) &= 1 - E(\hat{c}) \\ &= 1 - \gamma \\ &= \delta . \end{aligned}$$

Therefore $\hat{d}$ is unbiased.

$$\begin{aligned} \text{var}\ (\hat{d}) &= \text{var}\ (1-\hat{c}) \\ &= \text{var}\ (\hat{c}) \end{aligned}$$

$$\lim_{N\to\infty} \text{var}\ (\hat{d}) = \lim_{N\to\infty} \text{var}\ (\hat{c}) = 0 .$$

Therefore $\hat{d}$ is consistent.

DISTRIBUTION LIST

Navy

4 Dr. Marshall J. Farr, Director
Personnel & Training Research Programs
Office of Naval Research (Code 458)
Arlington, VA 22217

1 ONR Branch Office
495 Summer Street
Boston, MA 02210
Attn: Research Psychologist

1 ONR Branch Office
1030 East Green Street
Pasadena, CA 91101
Attn: E. E. Gloye

1 ONR Branch Office
536 South Clark Street
Chicago, IL 60605

1 Office of Naval Research
Area Office
207 West 24th Street
New York, NY 10011

6 Director
Naval Research Laboratory
Code 2627
Washington, DC 20390

12 Defense Documentation Center
Cameron Station, Building 5
5010 Duke Street
Alexandria, VA 22314

1 Special Assistant for Manpower
OASN (M&RA)
Pentagon, Room 4E794
Washington, DC 20350

1 LCDR Charles J. Theisen, Jr., MSC, USN
Code 4024
Naval Air Development Center
Warminster, PA 18974

1 Chief of Naval Reserve
Code 3055
New Orleans, LA 70146

1 Navy Personnel Research and
Development Center
Code 9041
San Diego, CA 92152
Attn: Dr. J. D. Fletcher

1 Dr. Lee Miller
Naval Air Systems Command
AIR-413E
Washington, DC 20361

1 Commanding Officer
U.S. Naval Amphibious School
Coronado, CA 92155

1 Chairman
Behavioral Science Department
Naval Command & Management
Division
U.S. Naval Academy
Luce Hall
Annapolis, MD 21402

1 Chief of Naval Education &
Training
Naval Air Station
Pensacola, FL 32508
Attn: CAPT Bruce Stone, USN

1 Mr. Arnold Rubinstein
Naval Material Command (NAVMAT
03424)
Room 820, Crystal Plaza #6
Washington, DC 20360

1 Commanding Officer
Naval Medical Neuropsychiatric
Research Unit
San Diego, CA 92152

1 Director, Navy Occupational Task
Analysis Program (NOTAP)
Navy Personnel Program Support
Activity
Building 1304, Bolling AFB
Washington, DC 20336

1 Dr. Richard J. Niehaus
Office of Civilian Manpower Management
Code 06A
Washington, DC 20390

1 Department of the Navy
Office of Civilian Manpower Management
Code 263
Washington, DC 20390

1 Chief of Naval Operations (OP-987E)
Department of the Navy
Washington, DC 20350

1 Superintendent
Naval Postgraduate School
Monterey, CA 93940
Attn: Library (Code 2124)

1 Commander, Navy Recruiting Command
4015 Wilson Boulevard
Arlington, VA 22203
Attn: Code 015

1 Mr. George N. Graine
Naval Ship Systems Command
SHIPS 047C12
Washington, DC 20362

1 Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054
Attn: Dr. Norman J. Kerr

1 Commanding Officer
Service School Command
U.S. Naval Training Center
San Diego, CA 92133
Attn: Code 3030

1 Dr. William L. Maloy
Principal Civilian Advisor for
Education & Training
Naval Training Command, Code 01A
Pensacola, FL 32508

1 Dr. Alfred F. Smode, Staff
Consultant
Training Analysis & Evaluation
Group
Naval Training Equipment Center
Code N-00T
Orlando, FA 32813

1 Dr. Hanns H. Wolff
Technical Director (Code N-2)
Naval Training Equipment Center
Orlando, FA 32813

1 Chief of Naval Training Support
Code N-21
Building 45
Naval Air Station
Pensacola, FL 32508

1 Dr. Robert French
Naval Undersea Center
San Diego, CA 92132

1 LCDR C. F. Logan, USN
F-14 Management System
COMFITAEWWINGPAC
Nas Miramar, CA 92145

1 Navy Personnel R&D Center
San Diego, CA 92152

5 Navy Personnel R&D Center
San Diego, CA 92152
Attn: Code 10

1 CAPT D. M. Gragg, MC, USN
Head, Educational Programs
Development Department
Naval Health Sciences Education
and Training Command
Bethesda, MD 20014

Army

1 Headquarters
U.S. Army Administration Center
Personnel Administration Combat
Development Activity
ATCP-HRO
Ft. Benjamin Harrison, IN 46249

1 Armed Forces Staff College
Norfolk, VA 23511
Attn: Library

1 Director of Research
U.S. Army Armor Human Research Unit
Building 2422, Morade Street
Fort Knox, KY 40121
Attn: Library

1 Commandant
United States Army Infantry School
Fort Benning, GA 31905
Attn: ATSH-DET

1 Deputy Commander
U.S. Army Institute of Administration
Fort Benjamin Harrison, IN 46216
Attn: EA

1 Dr. Frank J. Harris
U.S. Army Research Institute
1300 Wilson Boulevard
Arlington, VA 22209

1 Dr. Ralph Dusek
U.S. Army Research Institute
1300 Wilson Boulevard
Arlington, VA 22209

1 Mr. Edmund F. Fuchs
U.S. Army Research Institute
1300 Wilson Boulevard
Arlington, VA 22209

1 Dr. Leon H Nawrocki
U.S. Army Research Institute
1300 Wilson Boulevard
Arlington, VA 22209

1 Dr. J. E. Uhlaner, Technical
Director
U.S. Army Research Institute
1300 Wilson Boulevard
Arlington, VA 22209

1 Dr. Joseph Ward
U.S. Army Research Institute
1300 Wilson Boulevard
Arlington, VA 22209

1 HQ, USAREUR & 7th Army
ODCSOPS
USAREUR Director of GED
APO New York 09403

Air Force

1 Research Branch
AF/DPMYAR
Randolph AFB, TX 78148

1 Dr. G. A. Eckstrand (AFHRL/AS)
Wright-Patterson AFB
Ohio 45433

1 Dr. Ross L. Morgan (AFHRL/AST)
Wright-Patterson AFB
Ohio 45433

1 AFHRL/DOJN
Stop #63
Lackland AFB, TX 78236

1 Dr. Martin Rockway (AFHRL/TT)
Lowry AFB
Colorado 80230

1 Major P. J. DeLeo
Instructional Technology Branch
AF Human Resources Laboratory
Lowry AFB, CO 80230

1 AFOSR/NL
1400 Wilson Boulevard
Arlington, VA 22209

1 Commandant
USAF School of Aerospace Medicine
Aeromedical Library (SUL-4)
Brooks ARB, TX 78235

1 Dr. Sylvia R. Mayer (MCIT)
Headquarters, Electronic Systems Division
LG Hanscom Field
Bedford, MA 01730

1 CAPT Jack Thorpe, USAF
Flying Training Division (HRL)
Williams AFB, AZ 85224

1 AFHRL/PE
Stop #63
Lackland AFB, TX 78236

Marine Corps

1 Mr. E. A. Dover
Manpower Measurement Unit (Code MPI)
Arlington Annex, Room 2413
Arlington, VA 20380

1 Commandant of the Marine Corps
Headquarters, U.S. Marine Corps
Code MPI-20
Washington, DC 20380

1 Director, Office of Manpower Utilization
Headquarters, Marine Corps (Code MPU)
MCB (Building 2009)
Quantico, VA 22134

1 Dr. A. L. Slafkosky
Scientific Advisor (Code RD-1)
Headquarters, U.S. Marine Corps
Washington, DC 20380

1 Chief, Academic Department
Education Center
Marine Corps Development and Education Command
Marine Corps Base
Quantico, VA 22134

Coast Guard

1 Mr. Joseph J. Cowan, Chief
Psychological Research Branch (G-P-1/62)
U.S. Coast Guard Headquarters
Washington, DC 20590

Other DOD

1 Lt. Col. Henry L. Taylor, USAF
Military Assistant for Human Resources
OAD (E&LS) ODDR&E
Pentagon, Room 3D129
Washington, DC 20301

1 Mr. William J. Stormer
DOD Computer Institute
Washington Navy Yard, Bldg. 175
Washington, DC 20374

1 Col. Austin W. Kibler
Advanced Research Projects Agency
Human Resources Research Office
1400 Wilson Boulevard
Arlington, VA 22209

1 Dr. Harold F. O'Neil, Jr.
Advanced Research Projects Agency
Human Resources Research Office
1400 Wilson Boulevard, Room 625
Arlington, VA 22209

1 Helga L. Yeich
Advanced Research Projects Agency
Manpower Management Office
1400 Wilson Boulevard
Arlington, VA 22209

Other Government

1 Dr. Eric McWilliams, Program Manager
Technology and Systems, TIE
National Science Foundation
Washington, DC 20550

1 Dr. Andrew R. Molnar
Technological Innovations in Education
National Science Foundation
Washington, DC 20550

1 Dr. Marshall S. Smith
Asst. Acting Director
Program on Essential Skills
National Institute of Education
Brown Bldg., Room 815
19th and M St., N.W.
Washington, DC 20208

<u>Miscellaneous</u>

1 Dr. Scarvia B. Anderson
Educational Testing Service
17 Executive Park Drive, N.E.
Atlanta, GA 30329

1 Dr. John Annett
The Open University
Milton Keynes
Buckinghamshire
ENGLAND

1 Dr. Richard Snow
School of Education
Stanford University
Stanford, CA 94305

1 Dr. Gerald V. Barrett
Department of Psychology
University of Akron
Akron, OH 44325

1 Dr. Bernard M. Bass
University of Rochester
Management Research Center
Rochester, NY 14627

1 Dr. David G. Bowers
University of Michigan
Institute for Social Research
Ann Arbor, MI 48106

1 Mr. Kenneth M. Bromberg
Manager - Washington Operations
Information Concepts, Inc.
1701 North Fort Myer Drive
Arlington, VA 22209

1 Dr. Ronald P. Carver
School of Education
University of Missouri
Kansas City, MO 64110

1 Century Research Corporation
4113 Lee Highway
Arlington, VA 22207

1 Dr. Allan M. Collins
Bolt Beranek and Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. H. Peter Dachler
Department of Psychology
University of Maryland
College Park, MD 20742

1 Dr. René V. Dawis
Department of Psychology
University of Minnesota
Minneapolic, MN 55455

1 ERIC
Processing and Reference Facility
4833 Rugby Avenue
Bethesda, MD 20014

1 Dr. Victor Fields
Department of Psychology
Montgomery College
Rockville, MD 20850

1 Dr. Edwin A. Fleishman
American Institutes for Research
Foxhall Square
3301 New Mexico Avenue, N.W.
Washington, DC 20016

1 Dr. Ruth Day
Department of Psychology
Yale University
New Haven, CT 06520

1 Dr. Robert Glaser, Director
University of Pittsburgh
Learning Research & Development
Center
Pittsburgh, PA 15213

1 Dr. Robert Vineberg
HumRRO Western Division
27857 Berwick Drive
Carmel, CA 93921

1 LCol CRJ Lafleur, Director
Personnel Applied Research
National Defence HQ
Ottawa, Canada K1A 0K2

1 Dr. Henry J. Hamburger
School of Social Sciences
University of California
Irvine, CA 92664

1 Dr. M. D. Havron
Human Sciences Research, Inc.
7710 Old Spring House Road
West Gate Industrial Park
McLean, VA 22101

1 HumRRO
Division No. 3
P.O. Box 5787
Presidio of Monterey, CA 93940

1 HumRRO
Division No. 4, Infantry
P.O. Box 2086
Fort Benning, GA 31905

1 HumRRO
Division No. 5, Air Defense
P.O. Box 6057
Fort Bliss, TX 79916

1 Dr. Lawrence B. Johnson
Lawrence Johnson & Associates, Inc.
200 S. Street, N.W., Suite 502
Washington, DC 20009

1 Dr. Milton S. Katz
MITRE Corporation
Westgate Research Center
McLean, VA 22101

1 Dr. Steven W. Keele
Department of Psychology
University of Oregon
Eugene, OR 97403

1 Dr. David Klahr
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Ezra S. Krendel
Department of Operations Research
University of Pennsylvania
Philadelphia, PA 19104

1 Dr. Alma E. Lantz
University of Denver
Denver Research Institute
Industrial Economics Division
Denver, CO 80210

1 Dr. Robert R. Mackie
Human Factors Research, Inc.
6780 Cortona Drive
Santa Barbara Research Park
Goleta, CA 93017

1 Dr. Donald A. Norman
University of California, San Diego
Center for Human Information Processing
La Jolla, CA 92037

1 Mr. Brian McNally
Educational Testing Service
Princeton, NJ 08540

1 Mr. A. J. Pesch, President
Eclectech Associates, Inc.
P.O. Box 178
North Stonington, CT 06359

1 Mr. Luigi Petrullo
2431 North Edgewood Street
Arlington, VA 22207

1 Dr. Joseph W. Rigney
University of Southern California
Behavioral Technology Laboratories
3717 South Grand
Los Angeles, CA 90007

1 Dr. Leonard L. Rosenbaum, Chairman
Department of Psychology
Montgomery College
Rockville, MD 20850

1 Dr. George E. Rowland
Rowland and Company, Inc.
P.O. Box 61
Haddonfield, NJ 08033

1 Dr. Arthur I. Siegel
Applied Psychological Services
404 East Lancaster Avenue
Wayne, PA 19087

1 Dr. C. Harold Stone
1428 Virginia Avenue
Glendale, CA 91202

1 Mr. Dennis J. Sullivan
725 Benson Way
Thousand Oaks, CA 91360

1 Dr. Benton J. Underwood
Department of Psychology
Northwestern University
Evanston, IL 60201

1 Dr. David J. Weiss
Department of Psychology
University of Minnesota
Minneapolis, MN 55455

1 Dr. Anita West
Denver Research Institute
University of Denver
Denver, CO 80210

1 Dr. Kenneth N. Wexler
University of California
School of Social Sciences
Irvine, CA 92664

1 Dr. Arnold F. Kanarick
Honeywell, Inc.
2600 Ridge Parkway
Minneapolis, MN 55413

1 Dr. Carl R. Vest
Battelle Memorial Institute
Washington Operations
2030 M Street, NW
Washington, DC 20036

1 Dr. Roger A. Kaufman
United States International Univ.
Graduate School of Leadership and
Human Behavior
Elliott Campus
San Diego, CA 92124

(Continued from inside front cover)

211 J. Friend. Computer-assisted instruction in programming: A curriculum description. July 31, 1973.
212 S. A. Weyer. Fingerspelling by computer. August 17, 1973.
213 B. W. Searle, P. Lorton, Jr., and P. Suppes. Structural variables affecting CAI performance on arithmetic word problems of disadvantaged and deaf students. September 4, 1973.
214 P. Suppes, J. D. Fletcher, and M. Zanotti. Models of individual trajectories in computer-assisted instruction for deaf students. October 31, 1973.
215 J. D. Fletcher and M. H. Beard. Computer-assisted instruction in language arts for hearing-impaired students. October 31, 1973.
216 J. D. Fletcher. Transfer from alternative presentations of spelling patterns in initial reading. September 28, 1973.
217 P. Suppes, J. D. Fletcher, and M. Zanotti. Performance models of American Indian students on computer-assisted instruction in elementary mathematics. October 31, 1973.
218 J. Fiksel. A network-of-automata model for question-answering in semantic memory. October 31, 1973.
219 P. Suppes. The concept of obligation in the context of decision theory. (In J. Leach, R. Butts, and G. Pearce (Eds.), Science, decision and value. (Proceedings of the fifth University of Western Ontario philosophy colloquium, 1969.) Dordrecht: Reidel, 1973. Pp. 1-14.)
220 F. L. Rawson. Set-theoretical semantics for elementary mathematical language. November 7, 1973.
221 R. Schupbach. Toward a computer-based course in the history of the Russian literary language. December 31, 1973.
222 M. Beard, P. Lorton, B. W. Searle, and R. C. Atkinson. Comparison of student performance and attitude under three lesson-selection strategies in computer-assisted instruction. December 31, 1973.
223 D. G. Danforth, D. R. Rogosa, and P. Suppes. Learning models for real-time speech recognition. January 15, 1974.
224 M. R. Raugh and R. C. Atkinson. A mnemonic method for the acquisition of a second-language vocabulary. March 15, 1974.
225 P. Suppes. The semantics of children's language. (American Psychologist, 1974, 29, 103-114.)
226 P. Suppes and E. M. Gammon. Grammar and semantics of some six-year-old black children's noun phrases.
227 N. W. Smith. A question-answering system for elementary mathematics. April 19, 1974.
228 A. Barr, M. Beard, and R. C. Atkinson. A rationale and description of the BASIC instructional program. April 22, 1974.
229 P. Suppes. Congruence of meaning. (Proceedings and Addresses of the American Philosophical Association, 1973, 46, 21-38.)
230 P. Suppes. New foundations of objective probability: Axioms for propensities. (In P. Suppes, L. Henkin, Gr. C. Moisil, and A. Joja (Eds.), Logic, methodology, and philosophy of science IV: Proceedings of the fourth international congress for logic, methodology and philosophy of science, Bucharest, 1971. Amsterdam: North-Holland, 1973. Pp. 515-529.)
231 P. Suppes. The structure of theories and the analysis of data. (In F. Suppe (Ed.), The structure of scientific theories. Urbana, Ill.: University of Illinois Press, 1974. Pp. 267-283.)
232 P. Suppes. Popper's analysis of probability in quantum mechanics. (In P. A. Schilpp (Ed.), The philosophy of Karl Popper. Vol. 2. La Salle, Ill.: Open Court, 1974. Pp. 760-774.)
233 P. Suppes. The promise of universal higher education. (In S. Hook, P. Kurtz, and M. Todorovich (Eds.), The idea of a modern university. Buffalo, N. Y.: Prometheus Books, 1974. Pp. 21-32.)
234 P. Suppes. Cognition: A survey. (In J. A. Swets and L. L. Elliott (Eds.), Psychology and the handicapped child. Washington, D. C.: U. S. Government Printing Office, 1974.)
235 P. Suppes. The place of theory in educational research. (Educational Researcher, 1974, 3 (6), 3-10.)
236 V. R. Charrow. Deaf English--An investigation of the written English competence of deaf adolescents. September 30, 1974.
237 R. C. Atkinson and M. R. Raugh. An application of the mnemonic keyword method to the acquisition of a Russian vocabulary. October 4, 1974.
238 R. L. Smith, N. W. Smith, and F. L. Rawson. CONSTRUCT: In search of a theory of meaning. October 25, 1974.
239 A. Goldberg and P. Suppes. Computer-assisted instruction in elementary logic at the university level. November 8, 1974.
240 R. C. Atkinson. Adaptive instructional systems: Some attempts to optimize the learning process. November 20, 1974.
241 P. Suppes and W. Rottmayer. Automata. (In E. C. Carterette and M. P. Friedman (Eds.), Handbook of perception. Vol. 1. Historical and philosophical roots of perception. New York: Academic Press, 1974.)
242 P. Suppes. The essential but implicit role of modal concepts in science. (In R. S. Cohen and M. W. Wartofsky (Eds.), Boston studies in the philosophy of science, Vol. 20, K. F. Schaffner and R. S. Cohen (Eds.), PSA 1972, Proceedings of the 1972 biennial meeting of the Philosophy of Science Association, Synthese Library, Vol. 64. Dordrecht: Reidel, 1974.)
243 P. Suppes, M. Leveille, and R. L. Smith. Developmental models of a child's French syntax. December 4, 1974.
244 R. L. Breiger, S. A. Boorman, and P. Arabie. An algorithm for blocking relational data, with applications to social network analysis and comparison with multidimensional scaling. December 13, 1974.
245 P. Suppes. Aristotle's concept of matter and its relation to modern concepts of matter. (Synthese, 1974, 28, 27-50.)
246 P. Suppes. The axiomatic method in the empirical sciences. (In L. Henkin et al. (Eds.), Proceedings of the Tarski symposium, Proceedings of symposia in pure mathematics, 25. Providence, R. I.: American Mathematical Society, 1974.)
247 P. Suppes. The measurement of belief. (Journal of the Royal Statistical Society, Series B, 1974, 36, 160.)
248 R. Smith. TENEX SAIL. January 10, 1975.
249 J. O. Campbell, E. J. Lindsay, and R. C. Atkinson. Predicting reading achievement from measures available during computer-assisted instruction. January 20, 1975.
250 S. A. Weyer and A. B. Cannara. Children learning computer programming: Experiments with languages, curricula and programmable devices. January 27, 1975.

(Continued from inside back cover)

251 P. Suppes and M. Zanotti. Stochastic incompleteness of quantum mechanics. (Synthese, 1974, 29, 311-330.)

252 K. T. Wescourt and R. C. Atkinson. Fact retrieval processes in human memory. April 11, 1975.

253 P. G. Matthews and R. C. Atkinson. Verification of algebra step problems: A chronometric study of human problem solving. May 15, 1975.

254 A. F. Antolini. An investigation of the feasibility of computer-based generation of pattern drills for first-year Russian. June 23, 1975.

255 J. A. Van Campen and R. C. Schupbach. Computer-aided instruction in Old Church Slavic and the history of the Russian literary language. June 30, 1975.

256 M. R. Raugh, R. D. Schupbach, and R. C. Atkinson. Teaching a large Russian language vocabulary by the mnemonic keyword method. July 11, 1975.

257 J. Friend. Programs students write. July 25, 1975.