DOCUMENT RESUME

ED 275 316 IR 012 379

AUTHOR Fischer, Gwen Bredendieck
TITLE Computer Programming: A Formal Operational Task.
PUB DATE [86]
NOTE 16p.; Paper presented at the Annual Symposium of the Piaget Society (16th, Philadelphia, PA, 1986).
PUB TYPE Reports - Research/Technical (143) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Academic Achievement; *Cognitive Development; *Cognitive Style; Cognitive Tests; *Evaluation Criteria; Higher Education; Pretests Posttests; *Programing; Research Methodology; *Undergraduate Students
IDENTIFIERS *Piagetian Tests

ABSTRACT

Concerned with a high failure rate in computer programming courses, two studies were undertaken to discover if two individual cognitive styles--"analytic" (formal thought) and "heuristic" (concrete or pre-operational thought)--were predictors of performance in a beginning computer programming course. To appropriately measure those skills, a Piagetian-based paper and pencil test of cognitive development called "How Is Your Logic?" (HIYL) was administered to a total of 116 undergraduates in three beginning computer programming courses at a small liberal arts college. In the first study of 87 students, 91% who received a course grade of "B+" or higher were formal thinkers, while no one who was classified as concrete operational received a grade higher than a "C+." The correlation between course grades and HIYL was statistically significant. In study two, the performance of 29 students on a pre- and posttest of HIYL was compared with their performance in the course to measure cognitive development during the course; scores on the two forms did not demonstrate measurable development over the 10 weeks of the course. Using a Piagetian framework to analyze the text used by all classes, three major components requiring formal thought were identified--hierarchical classifications of abstract concepts, control structures, and top-down design--and each is discussed. The criteria for classification, weighting components, correlations between HIYL and grades, and correlations between course and test grades are displayed in tabular format. (DJR)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED275316

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

✓ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

Computer Programming: A Formal Operational Task

Gwen Bredendieck Fischer
Psychology Department
Hiram College
Hiram, Ohio 44234

Abstract

In Study 1, undergraduates in three beginning computer programming courses were given How Is Your Logic? (Gray, 1976) a Piagetian-based, paper and pencil test of concrete and formal operational thought. Ninety-one percent of the students who received course grades higher than B were formal operational; none who were classified as concrete operational received a course grade higher than C+. In study 2 performance on two forms of HIYL (administered as pre-test and post-test) were compared with performance in the course to measure cognitive development during the course. Results are explained by an analysis of the kinds of skills required in programming tasks and those measured by formal operational items on HIYL.

*The author wishes to express thanks to the beginning computer programming classes who voluntarily participated, to Rachel Taylor, Patricia Hugus, and Warne Edwards, who helped with scoring and data entry; to Dr. James Case, Dr. Janice Green, and Dr. Oberta Slotterbeck, for their supportive interest and giving me access to their classes and their tests; and to Dr. Mario Renzi for his help with SPSSx.

IR012379

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Gwen B. Fischer

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Computer Programming: A Formal Operational Task

Statement of the Problem

According to Hostetler (1983) and Mazlack (1980), computer programming courses are considered among the most difficult courses on many college campuses. As many as 15% - 25% of students enrolled in a programming class withdraw before the end of the course and an additional 10% - 15% receive failing or near failing grades. Concerned about this high failure rate, teachers and researchers have searched for individual differences which would predict success in programming courses. Most show little correlation. The best predictors appear to be G.P.A. and cognitive style. Of the two, G.P.A. is the less interesting because it fails to provide either a theoretically satisfying explanation for differences in programming ability or an intervention strategy. Cognitive style, on the other hand, appears more promising on both fronts.

Cheney (1980) identified two cognitive styles, which he called "analytic" and "heuristic". He characterized the former by capabilities which developmental psychologists associate with formal operational thought (that is, "model building, mathematical analysis, and optimization"); the latter by behaviors reminiscent of concrete operational, or even pre-operational thought (that is, "trial and error, ad hoc sensitivity analysis, muddling through, and . . selecting the first acceptable alternative"). Not surprisingly, Cheney found that analytic cognitive style was correlated with success on programming exams. Empirical support for the common sense hypothesis that programming requires logical analysis and hypothetical reasoning, suggests that an appropriate measure of those skills could be constructed on a Piagetian model.

The present series of studies were undertaken to discover whether a Piagetian-based test with high inter-rater reliability, internal consistency, and construct validity would be a good predictor of students' performance in a beginning computer programming course. In the first studies, I compared scores on How Is Your Logic?, (a Piagetian based test of cognitive development) with grades in three beginning programming classes. In the second study, students took two forms of How Is Your Logic? (HIYL), as pre- and post- tests, to measure whether they advanced in their level of thinking during the course. After reporting on the results of these studies, I will present an analysis of some of the specific programming tasks students are taught and measured on in the course, using a Piagetian framework to discover any structural similarity to formal operational tasks measured by HIYL.

Subjects and Procedure.

The first study consisted of 87 undergraduates enrolled in three classes of beginning computer programming at a small liberal arts college : Twenty-eight students were enrolled in in the winter of 1984 and 59 students were enrolled in two sections during the winter of 1985. Seventy-three percent were underclassmen. In the second study, an additional 29 student were enrolled in a section taught in the fall of 1985. All 116 students in both studies were given Form B of How Is Your Logic? (HIYL) (Gray, 1976) on a voluntary basis during the first week of the quarter. In addition, in study 2, students were also given Form A of <u>HIYL</u> ten weeks later (in the last week of classes). <u>HIYL</u> is a paper and pencil, group administered, Piagetian test. Both forms consist of 13 constructed-answer problems, 5 of which require concrete operations and 8 of which require formal operations. (See Gray, 1976 for a complete description.) See Table I for a description of the items requiring formal thought. The criteria used for classifying students as Concrete Operational (CO), Transitional (Tr), Early Formal Operational (FO I) or Consolidated Formal Operational (FO II) appear in Table I.

Students' success in the programming course was measured by their final course grades. All teachers involved in the series of studies base their course grades on a combination of out-of-class programming assignments and in-class tests. In Table II you can see that the weights they give the tests and the programming assignments differ considerably. I'll talk more about that when I report the findings. All <u>HIYL</u> tests were scored by the author and trained assistants with 87-90% agreement. The course instructors assigned grades with no knowledge of students' performance on HIYL; the scorers had no knowledge of students' performance in the computer course.

Results.

In the first study of 87 students, 91% of those who received a course grade of B+ or higher were classified as early formal (FO I) or consolidated formal thinkers (FO II) based on their performance on Form B of HIYL administered at the beginning of the course. No one who was classified as concrete operational received a course grade higher than C+. Of the 29 students classified as transitional, only 2 received a course grade higher than a B. The correlation between course grade and HIYL was statistically significant ($r = .62$, $p \leq .0005$).

When the study was replicated with a smaller sample of 29, the results were less compelling. As can be seen in Table III, the correlation between performance on pre-test Form B and course grade was .30 ($p \leq .06$) and between post-test Form A and course grade was .49 ($p \leq .004$). Because all three teachers agreed that students often work together on out-of-class programming assignments, making those grades perhaps less valid measures of individual ability, and because the 3 teachers weighted in-class tests and out-of-class assignments so differently (see Table II), I also compared performance on HIYL with test grades alone. Those correlations are slightly higher, somewhat more closely approximating those of the original study. As you can see in Table III, the correlation with Form B was .42 ($p \leq .012$) and with Form A, .51 ($p \leq .003$).

Comparing scores on the pre-test (Form B) and the post-test (Form A), does not support the hypothesis that a computer programming course might provide sufficient and appropriate disquilibrium to promote measurable development. Three students did score one level higher on post-test Form A than on pre-test Form B, the reverse was true for an additional 5 students. In explanation, some of my more skeptical colleagues have suggested that these findings provide support for their hypothesis that studying programming scrambles the brain, I prefer the more conservative explanation that ten weeks is too short a time over which to observe measurable developmental change and therefore the variation must be considred measurement error. An additional factor which may have contributed to the failure to detect developmental change is that, in the present study, no students were identified as concrete operational, and only 8 (28%) were identified as transitional, thus, there was little variability in performance on HIYL.

Analysis

Using a Piagetian framework to analyze the text (Pascal by Dale and Orshalick) used by all classes, I identified a number of components which require formal thought. I will discuss three major components: (1) Hierarchical classifications of abstract concepts, (2) control structures, and (3) top-down design.

To put these into the theoretical framework, let me briefly mention those characteristics (Piaget and Inhelder, 1955/1958) of formal thought which seem particularly relevant: (1) The ability to use combinatorial operations, (perceiving relationships as parts of a closely knit system within which the thinker can move easily). (2) The capacity for reasoning about inter-propositional operations so that reasoning becomes independent of factual content and focuses on establishing the logic between premises and conclusions; and the related ability (3) to separate form from content and to use symbols to represent statements whose truth and falisity is based on rules governing these inter-relations. (4) The understanding of a system of transformation in which the real or given can be compared with the probable and the merely possible.

Now, to look at programming tasks: First, hierarchical classification. The text introduces programming with a series of descriptions and definitions of concrete objects such as terminals, computers, main frames, disk drives, and printers. While representational (pre-operational) schemes might be sufficient to memorize appropriate definitions, the operation of classification is necessary to understand the relations among these objects. However, even though the referents for these terms are physically concrete, students cannot physically manipulate them to identify the effects of their presence or absence; thus, students might be able to successfully answer test questions requiring memorization of definitions, but be unable to answer those requiring understanding of the functioning of one part of the computer system with another. The next set of terms students must learn to recognize and categorize have even less manipulable referents---hardware, software, batches, inter-active systems and interfaces; from there the student is moved quickly to source, compiler, object programs, program listings, input data and program output, high- and low-level language, assembler and complier programs, modules, and machine language. A student who memorizes these terms, without understanding the relationships among their referents, may not only fail hypothetically posed questions about the functioning of parts of a computer system, but be reduced to random trial and error testing (or the help of a sympathetic, but more formal thinker) when confronted with the task of identifying the source of a program malfunction.

From there, matters only get worse: The student must learn yet another set of even more abstract terms. In order to understand the instructions for writing a program, the student must differentiate among characters, integers, and real numbers; variables, constants and values, reserved words, standard identifiers, and user names. A formal thinker, who is accustomed to thinking about structured wholes, will more easily be able to organize what must appear to the concrete thinker and novice a confusing mass of abstract terms---and in addition, the formal thinker will be able to understand , for example, that variables, constants, values, etc., are actually symbolic placeholders whose use is governed by rules which operate independently of factual reality or personal experience. Clearly second-order or formal reasoning.' If the programming teacher realizes that the concrete or transitional thinkers are able to use, but not create a systematic, but abstract classification system, then s/he may be able to aid the not-yet-formal thinker by the use of diagrams which show the hierarchical relationships among these various terms, as well as the functional relationships among them. Helping the concrete or transitional thinker understand the abstract, symbolic nature of variables and values may require many non-computer examples. Perhaps one reason science and mathematics majors are often more successful in computer courses than students with other majors is their familiarity with the use of such abstract concepts.

(2) A second component of Pascal programming requiring formal thought involves the use of various "control structures" such as looping and selection control. To use control structures, the programmer must be able to separate form from content, to understand the logic of the relations among propositions, as well as the closed nature of the system. When programming is first introduced, the order of the statements in the program corresponds to the order of the execution of the program. While the symbols and unfamiliar terms may create some problems for the beginning programmer, if the serial order of execution by the computer is represented by the order of statements, by imitating examples given in the text or in class, even a concrete or transitional thinker might be able to write simple programs. The correspondence between representation and execution is altered by the use of control structures.

Control structures direct the computer to make choices based on previous responses--for a program to do this, the "logical order of the program [must] differ from the physical order" (Dale and Orshalick, 1983). Thus, even given a problem with which the student has had direct experience, students can no longer rely on their direct personal experience in concrete problem-solving to provide the logic and rules which then are represented in computer language. Successful programming using control structures requires the separation of form from content and the ability to manipulate rules governing abstract propositions. For example, to construct a loop structure (regardless of the content of the problem), the programmer must understand how to start the loop and to stop it--i.e., must be able to recognize the circumstances under which the identity element is true (called "initializing the variable") and then be able to specify the negation of that variable--in order to avoid an infinite loop. Identifying the particular abstract element which fits this rule, and recognizing the need to specify the negation of that element requires that the programmer understand that the loop must constitute a closed system within the larger system of the program which must (by the rules of computer logic) contain both an identity element and its negation. The programmer must focus on the logic relating the individual propositions, regardless of the content of the particular progamming problem.

A selection control structure directs the computer to execute instructions conditionally. To use these structures the programmer must understand Boolean expressions, which requires understanding the relationship among variables, relational operators, and expressions; further, the programmer must understand the rules the computer will follow when given a variety of possible variables, expressions and operators. In addition, the rules specify that the values of the variables being operated on must come from the same category (e.g., within one expression, all values must be characters, real, <u>or</u> integer numbers); thus, a separate, but equally abstract classification system must be kept in mind.

The relational operators present another kind of problem for the novice programmer: Operators are common words ("AND", "OR", and "NOT") which have been given very precise definitions, presented in the form of truth tables such as are used in formal logic. The programmer must grasp what the computer will do given each of the possible statements, and this understanding requires both an understanding of truth tables as well as a suspension of the common sense translation of the expressions and the usual meaning of the operators.

(3) The third, but most central component of programming is what Dale and Orshalick call the "Top Down Design". They describe this method of programming as "working from the abstract (... description or specification of the problem) to the particular [or]...actual Pascal code) (p. 66)." They introduce this concept in a nice clear and concrete manner: How can you take the problem of giving a party and break it down into component parts (p. 68) ? Although most students will never have planned a party as consciously and systematically as Dale and Orshalick describe, none-the-less, at least parties are experiences with which most college students are familiar, and therefore this is a reasonable example to begin with. However, after that single example, the authors move on to examples which sound simple, but are problems with which students have had little or no first-hand experience, such as calculating weighted averages of a series of test scores.

Once the student can no longer rely on concrete experience, the process of breaking down a problem into component parts must entail formal thought: The student must be able to see a problem as a statement summarizing a set of inter-related, but unstated components, some of which are given, others probable, and still others are merely possible. Dale and Orshalick describe the process in the following way:

> We start by breaking the problem into a set of sub-problems. This process continues until each sub-problem cannot be further divided. We are creating a hierarchical structure, also known as a tree structure, of problems and sub-problems called functional modules. Modules at one level can call on the services of modules at a lower level. (p. 66)

To create such a tree structure, one must be able to conceive of all the possible steps or parts of each module or problem, and be able to determine in what order they must occur. When working with a problem with which the student has had concrete experience (such as giving a party), common sense and prior experience would tell a student that making a guest list must occur prior to locating their phone numbers. However, to find the weighted averages of a series of test scores, requires knowing how this process is done "by hand", otherwise, the tree structure in which the modules "get data" and "find average" are not helpful in determining what actual lines of programming must be written. Providing transitional thinkers with a great many examples using familiar content may show them the advantage of systematic analysis, while providing them with models for such analysis.

These three components of programming require the ability to reason about the relations among abstract and symbolic statements independent of factual content, and to use combinatorial operations in which relationships are seen as parts of a closely knit system and given instances are recognized as parts related to all possible instances.

Of the formal items on HIYL, the combinations and permutations problems would appear most likely to require the same capabilities. To check this analysis empirically, I grouped together those items on the two forms of HIYL which measure three different formal operations: (1) make correct inclusions, (2) deny incorrect inclusions, and (3) construct complete combinations and permutations. I then compared students' answers on these three composite variables---first with final course grade, and then with test grade. As you can see in Table IV, correlations of "construct complete combinations and permutations" with test grade was .57 ($p \leq .001$) and with test grade was .64 ($p \leq .0001$), suggesting that understanding the relationships of parts to wholes and systematically being able to construct all possibilities is perhaps the most critical operation for successful programming.

TABLE I

CRITERIA FOR CLASSIFICATION:

Eight items on How Is Your Logic? require the following formal operations:

Items Form A	Form B	
2, 3	9, 10	Making correct implications
5	6	Constructing complete combinations
6	7	Constructing systematic permutations
9, 10	3, 4	Denying incorrect implications
12, 13	11, 12	Proportional reasoning

These items could be solved at a preoperational, concrete operational, transitional, or formal operational level (a score of 7 or 8 indicates a formal answer and was required to be considered a successful solution).

LEVEL OF DEVELOPMENT:	CRITERIA
Consolidated Formal Thought (FO II)	= 6 - 8 Formal items solved
Beginning Formal Thought (FO I)	= 3 - 5 Formal items solved
Transitional (Tr)	= 1 - 2 Formal items solved
Concrete Thought (CO).	= None solved at the formal level

++

TABLE II

WEIGHTING OF COMPONENTS OF COURSE GRADES

	Date	Teacher	Basis for course grade
STUDY 1.			
	Winter, 1984 & Winter, 1985	1	out-of-class programming (15%), three in-class exams (15% each) in-class final exam (40%)
	Winter, 1985	2	out-of-class programming (33%) three in-class exams (33%) in-class final exam (33%)
STUDY 2.			
	Fall, 1986	3	out-of-class programming (50%) three in-class exams (12%, 13%, and 25%).

++

TABLE III

CORRELATIONS BETWEEN HIYL AND GRADES

STUDY 2.

	(pre-test) Form B	(post-test) Form A
Course Grade:	.2979 $p \leq .06$	.4944 $p \leq .004$
Test Grades:	.4175 $p \leq .012$	.5142 $p \leq .003$

CHARACTERISTICS OF FORMAL THOUGHT

Piaget and Inhelder (1966/1969 Psychology of the Child) identify five thought processes as marking the transformation from concrete operational thought to formal thought:

(1) **Reason about hypotheses and deduce logical consequences.** Essential to this capability is that the individual be able to reason from hypotheses or assumptions in which she does not believe. In other words, the individual must accept the hypothesis as tentative and subject to verification.

(2) **Reason about inter-propositional operations.** Reasoning independent of factual content---focusing on establishing the logic between premises and conclusion.

(3) **Separate form from content**---using algebraic or other symbols to represent statements in which truth or falsity is no longer based on factual reality, but on the rules governing the inter-relationships of statements within an argument.

(4) **Use of combinatorial operations.** The formal thinker perceives relationships as parts of a closely knit system, and so is able to move from one part of the system to another.

(5) **See the real as a subset of the possible.** Only when the subject understands the closed nature of the system of transformations can real instances be set into relationships and so compared with probable and the merely possible instances.

++

PROCESSES IN PASCAL PROGRAMMING REQUIRING FORMAL THOUGHT:

(1) **Hierarchical classification.** Terminals, computers, main frames, disk drives and printers. Hardware, software, batches, inter-active systems and interfaces; source, compiler, object programs, program listings, input data and program output; high-level language, low-level language, assembler and complier programs, modules, and machine language. Characters, integers, and real numbers; variables, constants and values; reserved words, standard identifiers, and user names.

(2) Separation of the serial order of the program statements from the serial order of the computer's execution of a program by control structures. The purpose of control structures is to provide a means by which the programmer can direct the computer to make choices based on previous responses, including starting the loop and stopping it--recognizing the circumstances under which identity element is true (called initializing the variable) and then specifying the negation of that variable--in order to avoid an infinite loop. Clearly a problem in reversibility.

(3) "Top Down Design". Beginning with a broad, abstract statement and breaking it down into component parts.

++

TABLE IV

CORRELATIONS:

		COURSE GRADE	TEST GRADE
CORRECT IMPLICATIONS		.229	.156
$p \leq .116$	$p \leq .207$		
DENY INCORRECT IMPLICATIONS		.351	.469
$p \leq .03$	$p \leq .005$		
COMBINATIONS /PERMUTATIONS		.573	.644
		$P \leq .001$	$P \leq .000$

References

Barker, Ricky J. and E.A. Unger. A Predictor for Success in an Introductory Programming Class Based Upon Abstract Reasoning Development. SIGCSE Bulletin. 15 (1) (1/83).

Cheney, Paul. Cognitive Style and Student Programming Ability: An Investigation. AEDS Journal. Summer, 1980.

Dale, and Orshalick. Pascal.

Gray, W. M. How Is Your Logic? (Experimental edition, Form A). Boulder: Biological Sciences Curriculum Study, 1976. (a)

Gray, W. M. How Is Your Logic? (Experimental edition, Form B). Boulder: Biological Sciences Curriculum Study, 1976. (b)

Hostetler, Terry R. Predicting Student Success in an Introductory Programming Course. SIGCSE Bulletin. 15 (3) (9/83).

Inhelder, B. and J. Piaget. The growth of logical thinkig from childhood to adolescence An essay on the construction of formal operational structures (A Parsons and S. Milgram, trans.). New York, Basic Books, 1958. (Originally published, 1955).

Mazlack, Lawrence J. Identifying Potential to Acquire Programming Skill. Communicationa of the ACM. 23 (1) Jan. 1980.

Note: ACM = Association for Computing Machines; SIGCSE = Special Interest Group on Computer Science Education.