A Discussion of Past Programming Errors and Their Effect on Learning Assembly Language

Kathleen M. Swigger*
North Texas State University, Denton, Texas

Layne F. Wallace
University of North Florida, Tallahassee, Florida

This study examines the question of whether it is possible to improve students' programming ability through repeated exposure to common programming errors. More specifically, it catalogs recurring errors in IBM 360/370 assembly language. The authors then use this information to test whether students who receive information and instruction concerning common errors have fewer programming errors than students receiving no such treatment. Results indicate that there is no significant difference between the two groups of students. These results seem to suggest that practice, more than instruction, is important in teaching students to find and correct programming errors.

INTRODUCTION

Although substantial research has focused on the area of teaching novices computer programming (see DuBoulay and O'Shea [1] for a complete literature review), much of the research on specific teaching methods lacks empirical evidence and remains a matter of personal preference. Coombs and Alty [2], for example, suggest that a student's early programming assignments concentrate on program command structures and input/output. Later exercises should build upon these initial assignments, adding variations to each new example. Weinberg [3] advocates program reading as a learning tool and laments the passage of the reading exercises that were once undertaken while programmers gathered to await their output. Bork [4] argues that novices should be taught using a "holistic" approach rather than a "grammatical" approach. The holistic approach exposes students to complete programs from the beginning and allows them to experience all programming constructs. The grammatical approach teaches about programming constructs in a bottom-up fashion. Lemos [5], however, compared the two approaches and found no significant difference between students' performances. Shneiderman [6] suggests that the two methods should be integrated into a single strategy called the "spiral" approach. Unfortunately, Shneiderman presents no empirical evidence to indicate the superiority of the spiral approach.

One of the areas of research on novice programmers that has achieved some results is that of identifying semantic networks for expert programmers [7] and showing how they relate to novices' understanding of programming [8]. There have been several studies on how knowledge networks are established by naive and novice programmers [9–12] and how these might be used to improve students' understanding of programming [13]. One of the most popular methods of gaining information about knowledge networks for novice programmers is to examine programming errors, which are the instances when the networks prove faulty [14]. Several attempts to catalog specific errors have been made [12, 15–18]. The knowledge gained from such studies has been used to construct intelligent tutoring systems [12, 19] that successfully diagnose students' programming errors. However, knowledge about student errors has never been used to enhance classroom experience, nor has it been used to test whether knowledge of such errors can prevent (rather than react to) students' programming errors.

The primary question asked in this paper is whether it is possible to improve students' programming ability by

Address correspondence to Dr. Kathleen M. Swigger, HRL/ID, Brooks Air Force Base, Brooks, Texas, 78235.

* This work was partially supported by HRL/ID, Brooks Air Force Base, San Antonio, Texas, in the form of Dr. Swigger's participation in the University Resident Research Program.

The Journal of Systems and Software 8, 395-399 (1988)

0164-1212/88/$3.50

teaching them about the common errors that occur in a programming language. Moreover, this study examines recurring errors in IBM 360/370 assembly language and divides these errors into two categories: misunderstood commands and semantic errors (a list of these errors is provided in Appendix 1). We then use this information to test whether students who are given instructions concerning common error types have fewer programming errors than students who receive no instruction concerning common error types.

Assembly language was selected as the test language because of its resemblance to the way the machine actually processes instructions. Previous research has indicated that students using a high-level language such as Pascal, Basic, or PL/I do not really understand the proper use or implication of some of the language's statements [12, 16–18]. Some of this misunderstanding may be due to the students' lack of a notion of a machine. Research, for example, indicates that beginners develop a conceptual model as they learn to use a computer system [20] and that there are benefits in providing an explicit model of a computer for novices learning a programming language [21, 22]. Thus, assembler might be a better place to start in examination of students' errors and in deciding whether knowledge of common errors really improves programming ability. In assembly language, each command causes one or at most two changes, and each change may be described and understood in concrete terms. Thus, assembly language forces students to explain their knowledge of the programming language as well as their misunderstandings in very concrete terms.

TEST I. AN EXAMINATION OF PROGRAMMING ERRORS

Method

The first study cataloged common error types occurring in assembly language. The subjects were graduate and undergraduate students attending North Texas State University and enrolled in an introductory IBM 360/370 assembly language course. Students enrolled in this class have previously had programming courses, one in Basic and one in Pascal. However, the Basic course is actually an overview of computer science applications with only three weeks devoted to the study of the Basic language. Therefore, after much consultation, it was determined that students enrolled in this course should still be classified as "novice" programmers. Subjects of our study were taken from these courses for a period of three extended semesters. The effect of an individual instructor's teaching style was reduced by using three different instructors to counterbalance styles. While the textbook for each class was the same, all instructors used a variety of sources for their lecture material. To reduce the possibility of having errors specific to a particular programming assignment, all instructors gave different assignments throughout the semester. To ensure complete independence of results, the instructors collected data from only their own classes for the three semesters and then collated the results at the end of the research period.

Collection of data was accomplished by observing which errors occurred repeatedly during debugging sessions with the students. Care was taken to make sure that a single student was the source of all errors. The errors were then divided into misunderstood commands and semantic categories.

Results and Discussion

It was found that the same programming errors did indeed appear across classes and across semesters. Students who made these errors did not seem to fall into groups divided by sex, age, or education level. One of the common factors, however, was the amount of exposure to the concepts used in assembly language programming at an intimate level—in writing programs, for example. After students were made aware of the faulty concept, they did not usually commit that specific error again, or if they did, they were able to recognize the error and correct it without assistance from the instructor.

While some questions were catagorized as misunderstood commands, the underlying cause may actually have been semantic. For example, one error found frequently was to use a DS statement instead of the DC statement to initialize a variable. This seems to be purely a faulty network node concerning the use of DS and DC and their implications for computer consequences. Another example of a faulty network is the error of trying to MVC data to an output file instead of using the PUT macro. The previous example may be an indication that the networks for high-level languages are organized in a different fashion than the networks for assembly language. An alternative explanation could be that of cognitive interference, where the concept of printing in assembly language has become confused with the concept of printing in a high-level language.

The errors we found to be the most common were presented to other assembly language teachers who had taught this course in the past using different textbooks, and all agreed that these errors seemed to be the most prevalent. This provides some continuity between the present instructors and the past instructors, as well as indicating that the errors are not textbook-specific. The

question of whether these findings can be generalized to other universities has yet to be answered.

Finally, it was discovered that students, as a group and across semesters, continue to make mistakes even after the instructor has repeatedly cautioned the class against making such errors. This leads one to theorize that students do not process some information at a practical level during lecture but when forced to assimilate information while writing a program they tend to retain the concepts. Findings such as these suggest that only practical experience with a language facilitates the establishment of valid networks. Since the initial research was obviously done by naturalistic observation, the next logical step was to replicate these observations using more experimental techniques.

TEST II. EFFECT OF INSTRUCTION ON ERRORS

Method

The second study looked at whether it is possible to imrpove students' programming performance if they are instructed about error types. Seventy computer science students were the subjects in this second experiment. All were either graduate or undergraduate students enrolled in an undergraduate course in assembly language programming. Each had previous programming experience in at least Pascal and Basic (one course in Pascal and one course in Basic). But again the Basic course consisted of an overview of computer applications with some emphasis on Basic. Students enrolled in the assembly language course had had only one real computer science course. All the students who participated in this study were computer science majors.

The subjects were divided into two groups, each enrolled in one section of the course. Both classes were taught on the same day, by the same instructor, one hour apart. The first class was given a list of errors (see Appendix 1) and was instructed on how to avoid specific errors and how to correct errors. Each error was discussed in detail in classroom lectures. The second class received no list of errors and was not lectured on error types or specific errors. Instead, the students in the control group received additional examples on each instruction. All subjects were given the same homework and programming assignments, and both were instructed and tested on how to read memory dumps.

A short program (less than 100 lines—see Appendix 2) was assigned to the two groups. The programming assignment was not logically complicated. No students had difficulty with the overall algorithm, although many experienced difficulties with small details. Collection of data was accomplished by having students route all output from execution of the programs to the instructor's file area.

Students' programming errors were grouped into six categories corresponding to the error types previously mentioned. Syntax errors flagged by the assembler were not considered in this study. Furthermore, only categories that contained a total of at least five errors were included in the comparison. If a student submitted multiple copies of the same error, the error was counted as a single error. A chi-square test was performed to determine whether there were any significant differences between the two groups in any of the six categories.

Results and Discussion

It was expected that the group who received extensive instruction regarding assembly language errors would have fewer errors and dumps than the group receiving no instruction. However, no significant differences were found between the groups for any of the six categories ($x = 9.73$, df $= 5$, $P < 0.08$). At most, we can say that there is a trend indicating that some treatment may help alleviate errors, especially for errors relating to the category labeled Declaration errors. Although the treatment group (Table 1) actually had fewer dumps than the group receiving no treatment, this was not significant.

An analysis of the specific categories reveals that, even into the third course, students continue to have difficulty understanding storage and the manipulation of storage. The categories dealing with storage declaration and storage transfer accounted for over 50% of all errors. Such a phenomenon suggests that even students in their third programming course do not have a firm notion of how storage is manipulated inside a computer. Although the curriculum in the first and second courses includes a discussion of addresses, storage, and memory locations, this specific content is obviously not fully understood by most of the students. The question of whether this is a unique phenomenon of this study or something that is generalizable needs further testing.

CONCLUSION

Unfortunately, evidence presented in this study indicates that instruction on error types using this particular method has no significant effect on programmers'

Table 1. Number of Errors for Each Treatment

Error type	No treatment	Treatment
Declaration/initialization	37	14
Index/address	9	14
Loop	8	7
Storage transfer	22	23
Input	11	11
Accumulation	6	6

performance. Apparently, some exposure to the machine and the errors in the context of a machine may be an integral part of forming an accurate and usable mental model of programming. Even if learning, thinking, and debugging strategies, whether general or specific, are shown to exist, it might not be possible to teach them directly. Perhaps they must spontaneously emerge as a consequence of substantial experience. At the very least, it might be possible to select and design experiences to result in a more rapid and complete emergence of such skills. These findings suggest that courses that try to teach a programming language using in-class exercises alone are inadequate for knowledge transfer to occur. Taking this idea a step further, programming courses that require the student to write programs and then run these programs may convey information at a level that will allow prolonged retention of the material.

Future research should be directed toward examination of the actual knowledge networks being used by beginning students as well as the manner in which the students build such networks. This study suggests that class exercises about possible error types are not sufficient to eradicate student programming errors and are perhaps insufficient for building needed knowledge nets. Research should be initiated to determine whether experience or prolonged exposure to a construct will help formalize an internal model that can avoid common error types. We are currently examining the impact of experiences on error types and are cataloging errors that occur over time. Individual student's errors are being mapped over time to determine if student error patterns exist. Such studies should aid in both the teaching and understanding of programming.

APPENDIX 1. MOST COMMON PROGRAMMING ERRORS

Misunderstood Commands

Using a DS instead of a DC to initialize a variable

Using L instead of CVB and/or ST instead of CVD

Trying to access the results of a D or DR as though it were one number

Using an incorrect EDit pattern and getting incorrect (or no) output

Not realizing the implications that MVC uses the length of the first operand to determine the number of bytes to be moved

Semantic Errors

Using MVC to print data

Reserving the incorrect number of bytes for storage with DS and DC statements

Trying to propagate blanks through the print area without defining a blank immediately before the print area

Trying to do packed arithmetic with an invalid sign in one or both operands

Trying to print numeric data without changing the sign portion to character format

Confusing address manipulation with data manipulation

Forgetting that the number system for IBM 360/370 assembly language starts with zero and not one

Inability to distinguish execution-time operations from compile-time operations

Forgetting to initialize registers or memory locations

APPENDIX 2. PROGRAM DESCRIPTION

Given a list of 20 employees, along with their ID numbers and monthly payroll figures, print *one* list of employees who make under $2500 a month and *one* list of employees who make $2500 or over a month. Each list should contain the names, employees IDs, and payroll for the individuals and a *total* for that group. On a separate line, print the *total* for both groups (*all* employees).

REFERENCES

1. B. DuBoulay and T. O'Shea, Teaching Novices Programming, in *Computing Skills and User Interface* (M. Coombs and J. Alty, Eds.), Academic, London, 1981, pp. 147–200.
2. M. Coombs and J. Alty, Eds., *Computing Skills and User Interface*, Academic Press, London, 1981, pp. 145.
3. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand-Reinhold, New York, 1971.
4. A. M. Bork, Learning to Program for the Science Student, *J. Educ. Data Proc.* 8, 1–5 (1971).
5. R. S. Lemos, Fortran Programming: An Analysis of Pedagogical Alternatives, *J. Educ. Data Proc.* 12(3), 21–29 (1975).
6. B. Shneiderman, Teaching Programming: A Spiral Approach to Syntax and Semantics, *Comp. Educ.* 1, 193–197 (1977).
7. K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle, Knowledge Organization and Skill Differences in Computer Programmers, *Cognitive Psychol.* 13, 307–325 (1980).
8. H. Kaheney, Problem Solving by Novice Programmers, in *The Psychology of Computer Use* (T. R. Green, S. J. Payne, and G. C. van der Veer, Eds.), Academic, London, 1983, pp. 121–142.
9. Friedman, D. *The Little Lisper*. Science Research Associates, Calif., 1974.
10. J. T. Schwartz, What Programmers Should Know, *Comput. Lang.* 2, 25–39.
11. N. Wirth, The Programming Language Pascal, *Acta Inf.* 1(1), 35–63 (1977).
12. E. Soloway, J. Bonar, B. Woolf, E. Barth, and K. Erlich, Cognition and Programming: Why Your Students Write

Those Crazy Programs. *Proc. Nat. Educ. Computer Conf.*, Denton, Texas, 1981, pp. 206–219.
13. J. Hoc, Analysis of Beginners' Problem-Solving Strategies in Programming, in *The Psychology of Computer Use* (T. R. Green, S. J. Payne, and G. C. van der Veer, Eds.), Academic Press, London, 1983, pp. 97–127.
14. J. D. Gannon, Characteristic Errors in Programming Languages, *Proc. ACM Ann. Conf.*, 1978, pp. 75–81.
15. D. Sleeman, R. T. Putnam, J. Baxter, and L. Kuspa, Pascal and High School Students: A Study of Errors, *J. Educ. Comput. Res.* 2(1), 5–24 (1986).
16. K. M. Swigger and F. L. Wallace, Use of Appropriate Looping Structures: Expert vs. Novice. *Proc. Human Factors Society,* Seattle, Washington, 1982, pp. 999–1003.
17. P. Bayman and R. E. Mayer, A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements, *Commun. ACM* 26(9), 677–679 (1983).
18. S. R. Rulon, Beginners' Misconceptions of Nine BASIC Statements. *Proc. Fed. North Texas Area Universities,* Denton, Texas, 1984, pp. 84–92.
19. J. Anderson and B. Reiser, The LISP Tutor, *Byte* 10(4), 159–178 (1985).
20. T. P. Moran, An Applied Psychology of the User, *Comput. Surv.* 13(1), 1–11 (1981).
21. R. E. Mayer, The Psychology of Learning BASIC, *Commun. ACM* 22(11), 589–493 (1979).
22. R. E. Mayer, The Psychology of How Novices Learn Computer Programming, *Comput. Surv.* 13(1), 121–141 (1981).