EXPERIMENTAL EVALUATION OF PROGRAMMING LANGUAGE FEATURES: IMPLICATIONS FOR INTRODUCTORY PROGRAMMING LANGUAGES

Richard Furuta

Computer Science, FR-35
University of Washington
Seattle, Washington 98195

P. Michael Kemp

BNR Inc.
Stanford Industrial Park
3174 Porter Drive
Palo Alto, California 94304

Abstract

The interaction between programmer and programming language affects the maintainability, reliability, and understandability of the resulting programs. Their results are important both to the educator and to the language designer, particularly when examining languages to be used when teaching beginning programmers. The purpose of this paper is to examine and discuss the methodology and results of a number of these research studies.

Introduction

As computer applications become more complex, the need for programming styles which result in understandable, reliable, and maintainable programs becomes more evident. Often, however, programmers are unaware of the equally important ways in which one's programming language guides and constrains one's actions, thereby strongly influencing the quality of the programs written in that language. Indeed, it appears that one of the more important factors in the production of maintainable, modifiable software is the interaction between programmer and programming language.

Several human factors researchers have studied this interaction, specifically in the area of design of programming languages, to attempt to determine empirically which language features support reliable, high quality software. Unfortunately for the designers of higher level languages, much of this research has been performed with naive or semi-naive subjects which raises questions about the applicability of the results to professional programmers. However, the educator is more fortunate since this empirical evidence is directly applicable to the choices involved in selection or design of programming languages for beginning students. The purpose of this paper is to discuss the applicability of human factors research to programming languages for beginners.

R.F. wishes to acknowledge a grant from the Graduate School at the University of Washington. We also wish to thank Steve Hedetniemi, Chairman of the Computer Science Dept. at the University of Oregon, and Andrew Coward of BNR Inc., for support which made this effort possible. Our special thanks to those colleagues who contributed their insight and advice.

Research in this field can be divided into two major catagories. One is the comparison of programming language constructs. The first is more useful to the educator choosing between several languages, the second to the educational language designer. Before looking at the results of these studies, it is important to examine a number of methodological considerations which can affect the applicability and usefulness of the results.

Methodological Considerations

There are a number of problems involved in experimental research of programming languages. Highly controlled experiments provide the most accurate and reliable results, but are the most difficult to generalize.

Comparison of two programming languages can be useful in making decisions about the adoption of one language as opposed to another in regard to certain specified goals, e.g., reliability or readability. However, this approach provides no insight as to why one language performs better than another. Comparison of individual language features in a carefully controlled experiment provides the most reliable information as to which features contribute the most toward the specified goal. Unfortunately, it is difficult to generalize beyond the conditions of the particular experiment.

A major area of controversy involves the use of naive subjects as opposed to experienced programmers. Some tasks require use of experienced programmers, and since experienced programmers in general are paid for their time, and in general have other conflicting assignments, the experiment cannot be designed to use a large number of subjects (to reduce the effect of individual differences) nor, in most cases, can it be designed to involve larger programming problems. Furthermore, it is difficult to quantify the background of the individual programmers involved.

The majority of experiments are carried out with naive subjects, primarily because naive subjects are easy to obtain in a university environment. Whether the use of naive subjects affects the generality of the results is an open question but some evidence indicates that it does not. Results obtained with naive subjects have been duplicated when using professional programmers as subjects [GREE77].

In any case, results from experiments with naive subjects clearly allow us to assign values to constructs found in programming languages which are used in introductory courses.

Unfortunately, large differences in the quality of the programs produced by programmers rated to be of "equal" skill makes any analysis difficult, for example, see [RUBE68].

Additionally, as Weinberg has demonstrated [WEIN74], the choice of a goal for a programming project (for example, whether it is more important to complete the project quickly or to have the programs run in the least amount of time) affects the programs produced. When the goals of an experiment or project are not clearly stated (and in benchmark projects there are generally many conflicting goals to be reached), the programmers involved have an almost infinite number of choices of which goals to strive for, thus making the validity of the results difficult to prove. It is probable that some of the differing language features cause improvement and others cause deterioration, but it is not possible to say which features have which effect. This is not as major a drawback to the person attempting to choose between languages as it is to the person attempting to design a language. One solution is to simplify the problem by restricting the size of the languages under study and by restricting the differences between them. In fact, much of the experimental work done has been performed with microlanguages which differ in only one (or in a very few) constructs.

Comparing Several Programming Languages

Frequently, the educator must decide which of several available programming languages is best suited to a particular programming class. Many factors must be considered: cost, the language implementation, and the specific application. In order to determine which of several programming languages is most suitable for a given application, an experiment may be designed to compare the languages under consideration.

A common approach to evaluate several languages is to code a benchmark problem in each of the languages under consideration, and then to compare the resulting programs with certain criteria (typically, speed of execution, programmer time to completion of the program, etc). Rubey [RUBE68] did just that in a loosely controlled experiment comparing PL/I with FORTRAN, COBOL, and JOVIAL. He concluded that PL/I was a usable language, although non-professional programmers might be better off using a more specialized, simpler language. He also indicated that his results were difficult to interpret in more detail since the differences in ability between programmers were found to be greater than the differences created by the languages.

Nutt [NUTT78] published an informal comparison of FORTRAN and PASCAL based on the teaching of two similar introductory Computer Science courses, one using FORTRAN, the other using PASCAL. He indicates that students learning PASCAL seem to learn better programming techniques than do those who learn FORTRAN. Additionally, it appears that it is easier for students who have learned PASCAL to learn FORTRAN than it is for students who have learned FORTRAN to learn PASCAL. Unfortunately, this study is largely uncontrolled. Although the programming courses involved are similar, there are more substantial differences between them than just the programming languages used. It is easy, therefore, to argue that the results apply only to the one specific situation reported.

The aforementioned studies illustrate one of the major difficulties in comparing two or more general purpose languages: The languages are so rich and the measurement techniques so broad that it is impossible to say much more than that one language (as a whole) appears to be better than the other.

Reisner has successfully compared two complete languages [REIS75, REIS77]. Her research compared two database query languages, SQUARE (an existing language), and SEQUEL (a language under development). Subjects in the experiments were representative of the expected naive users of the languages. She was able to assess strengths and weaknesses of both languages as well as to make specific set of porblems, and since the experiment was well controlled.

Comparing Individual Language Constructs

The educational language designer faces a different task. Again, implementation, cost, and applicability are important factors. Also important is the determination of which features will enhance and which will degrade program readability, maintainability, and reliability. The designer can turn to research which compares individual language constructs. Some of the results are summarized below:

Sime, et al. [SIME73, SIME77a] have extensively studied the conditional branching statement. In their experiments, naive subjects construct short programs in one of several microlanguages differing from each other only in the branching structure used. They have found that a redundant encoding of the branch condition resulted in the lowest error lifetimes:

IF <cond> <act1> NOT <cond> <act2> END <cond>
where <cond> is the boolean expression and <act1> , <act2>, and <act3> are actions to be performed conditionally on the boolean expression. Second best was the ALGOL-like nested IF:

IF <cond> BEGIN <act1> END ELSE BEGIN <act2> END.
and worst was the BASIC-like form:

IF <cond> GO TO <label>

Experiments by Green indicate than these results also hold true for professional programmers [GREE77]. Sime, et al., also suggest a writing procedure [SIME77b, SIME77c] in an attempt to reduce the number of careless syntactic errors committed while trying to write conditional branch statements of the second form (most commonly, leaving off a BEGIN or its matching END). Instead of selecting individual works and composing entire phrases from them, programmers are advised to start by selecting an entire phrase and then making substitutions into the phrase at specified places.

In work complementary to that of Sime, et al., Mayer [MAYE76b] reports that programs with the IF <cond> GO TO <label> conditional branch used freely are more difficult for naive users to understand than are programs with this construct arranged so that the jumps are as short as possible. For more difficult problems, programs written using the ALGOL-like nested IF are more readily understood than those using the short jumps (for smaller problems, the two are about the same). Interestingly, the most understandable form was one in which possible conditions were expressed as a contingency table with the results expressed on the same line as the necessary conditions (similar in form to the CASE statement). Mayer is an educational psychologist and some of his earlier works report on teaching a subset of FORTRAN to beginners with and without using a physical model of the computer [MAYE75, MAYE76a]. The physical model, called the Activator, aided the students in the initial learning phase.

Gannon has presented several intriguing experiments. His subject choice is unusual in that he uses relatively advanced (senior and graduate level) students rather than naive subjects. He has made a distinction between error frequency (how often the error occurs) and error persistence (how long it takes to correct the error once it occurs). In his first set of experiments [GANN75a, GANN75b, GANN76a] he reports that use of the semicolon as a statement terminator (as in PL/I) rather than as a statement separator (as in ALGOL and PASCAL) results in an order of magnitude decrease in error frequency although error persistence remains the same. Additionally, errors resulting from confusion between the assignment operator, ":=" , and the equality operator "=" are of great persistence. He suggests replacing ":=" with a backarrow in language designs. In more recent research [GANN76b, GANN77], Gannon has studied the method by which a language associates data types with operands. He finds that to associate types with operands statically at compile time (as in PADCAL) is less error prone than it is to associate types with operands dynamically at execution time (as in SNOBOL). As might be expected from these results, typeless languages which treat all operands as vectors of bits (for example, BLISS) tend to be the most error prone of all.

Well designed experiments of the above type provide the language designer with well controlled results. However, these results are quite specific to the conditions of the experiment. They must be examined carefully to determine if the results can be generalized to the desired environment and application. In many cases, further experimentation will be required to insure proper application of the results.

Discussion and Conclusions

We have examined two types of studies: the comparison of entire languages, and the comparison of language constructs. The educator and designer may look at both during the decision process.

The comparison of languages offers great promise in the area of choosing languages for specific applications. A good possibility for future research would be to combine the methodology of Reisner with the goals of Nutt in examining the relative worth of languages such as FORTRAN and ALGOL for use in introductory courses.

Perhaps the brightest future lies in the comparison of individual features. It is only by varying one factor at a time that the most scientific and replicable results are obtained. Several problems remain to be solved, however. Further research is clearly needed to determine the relative suitability of naive and experienced subjects. Careful attention also needs to be given to the measurement criteria. Additionally, assumptions about applications should be clearly stated. As the field expands, however, more and more features in each area of application will have been tested against similar criteria. The interaction of these "good" features will then require careful study. The most promising features should then be combined into microlanguages, and the interaction of desirable features can be measured. This is necessary to ensure that the effects of the individual features do not cancel each other and that the resulting language is a blend of features which all contribute positively toward a given goal.

Certainly there are many factors involved in the choice or design of an educationally useful programming language, and the research cited here stimulates new questions as it provides answers. Nevertheless, this methodology provides a way to supply many of these answers in a scientific manner.

Further References

Besides those works cited above, the reader is referred to the excellent comprehensive survey by Miller and Thomas [MILL77], to the annual annotated bibliography on computer program engineering compiled at the University of Toronto (most recently [BARN771]),and to our own bibliography [FURU79].

Additionally, Shneiderman [SHNE76a, SHNE76b, SHNE77a, SHNE77b] presents results dealing more with questions of program quality than with programming constructs. His topics include commenting, modularity, variable name selection, listing indentation, and flowcharting. Myers [MYER78] has explored the area of structured walkthroughs and testing.

Finally, a number of empirical studies have been performed which attempt to characterize a language or its users by inspection of the statement distributions, error frequencies, or other artifacts of the programming process found in a sample of programs taken from the population under study. For example, [MOUL67] and [KNUT71] report on statement distributions in FORTRAN, [LITE76] tabulates errors made by students learning COBOL, [ENDR73]describes errors found during a large project using assembly language, and [ELSH76]analyzes the style used in PL/I programs produced by professionals. These reports provide an instructor with a supplement to his intuition.

Specifically, they can warn him about possible problem areas in a language which has been selected for use in a class. As well, these reports help the developer of experiments comparing microlanguages to decide which constructs to test first, since he can see which constructs are used most often or are particularly error prone. Generally, however, these reports do not provide a basis for comparing one computer language to another, and are therefore not helpful in solving the problem of determining relative merit.

Additionally, since data collection is rarely performed under controlled conditions in studies of this type, it is easy to question the applicability of reported data to situations other than the specific ones involved.

References

BARN 77 Barnard, D., editor. An annotated bibliography on computer program engineering, Fifth edition. University of Toronto Technical Report CSRG-80. (May 1977).

ELSH76 Elshoff, J.L. An analysis of some commercial PL/I programs. IEEE Transactions on Software Engineering SE-2, 2 (1976), pp. 113-120.

ENDR73 Endres, A. An analysis of errors and their causes in systems programs. IEEE Transactions on Software Engineering SE-1,2 (1973), pp. 140-149.

FURU79 Furuta, R. and Kemp, P. M. Experimental evaluation of programming language constructs and related references: A bibliography including summaries. University of Oregon Technical Report (in preparation).

GANN75a Gannon, J.D., and Horning, J.J. Language design for programming reliability. IEEE Transactions on Software Engineering SE-1, 2 (1975), pp. 179-191.

GANN75b Gannon, J.D., and Horning, J.J. The impact of language design on the production of reliable software. Proceedings of the International Conference on Reliable Software (1975).

GANN76a Gannon, J.D. An experiment for the evaluation of language features. International Journal of Man-Machine Studies 8(1976), pp. 61-73.

GANN76b Gannon, J.D. Data types and programming reliability: Some preliminary evidence. Presented at the Symposium on Computer Software Engineering, Polytechnic Institute of New York (April 20-22, 1976).

GANN77 Gannon, J.D. An experimental evaluation of data type conventions. Communications of the ACM 20,8(1977), pp. 584-595.

GREE77 Green, T.R.G. Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology 50(1977),pp.93-109.

KNUT71 Knuth, D.E. An empirical study of FORTRAN programs. Software -- Practice and Experience 1,2(1971), pp. 105-133

LITE76 Litecky, C.R. and Davis, G.B. A study of errors, error-proneness, and error diagnosis in COBOL. Communications of the ACM 19,1(1976), pp. 33-37.

MAYE75 Mayer, R.E. Different problem-solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology 67,6(1975), pp. 725-734.

MAYE76a Mayer, R.E. Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. Journal of Educational Psychology 68,2 (1976),pp. 143-150.

MAYE76b Mayer, R.E. Comprehension as affected by structure of problem representation. Memory & Cognition 4,3(1976),pp. 249-255.

MYER77 Myers, G.J. A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM 21, 9(1978), pp. 760-768.

MILL77 Miller, L.A. and Thomas, J.C., Jr. Behavioral issues in the use of interactive systems. International Journal of Man-Machine Studies 9(1977), pp. 509-536.

MOUL67 Moulton, P.G. and Muller, M.E. DITRAN-A compiler emphasizing diagnostics. Communications of the ACM 10,1(1967), pp. 45-52.

NUTT78 Nutt, G.J. A comparison of PASCAL and FORTRAN as introductory programming languages. SIGPLAN Notices 13,2(Feb.1978), pp. 57-62.

REIS75 Reisner P., Boyce, R.F., and Chamberlin, D.D. Human factors evaluation of two data base query languages -- SQUARE and SEQUEL. Proc. AEIPS National Computer Conference 44(1975), pp. 447-452.

REIS77 Reisner, P. Use of psychological experimentation as an aid to development of a query language. IEEE Transactions on Software Engineering SE-3,3(1977), pp. 218-229.

RUBE68 Rubey, R.J. A comparative evaluation of PL/I. Datamation 14,12(Dec 1968), pp. 22-25.

SHNE76a Shneiderman, B. Exploratory experiments in programmer behavior. International Journal of Computer and Information Sciences 5,2(1976), pp. 123-143.

SHNE76b Shneiderman, B.& McKay,D. Experimental investigations of computer program debugging and modification. Presented at the 6th International Congress of the International Ergonomics Association, College Park Md. (July 1976).

SHNE77a Shneiderman, B., Mayer, R., McKay, D., and Heller, P. Experimental investigations of the utility of detailed flowcharts in programming. Communications of the ACM 20,6(1977), pp. 373-381.

SHNE77b Shneiderman, B. Measuring computer program quality and comprehension. International Journal of Man-Machine Studies 9(1977), pp. 1-14.

SIME73 Sime, M.E., Green, T.R.G., and Guest,D.J. Psychological evaluation of two conditional constructions used in computer languages.International Journal of Man-Machine Studies 5,1(1973),pp. 105-113.

SIME77a Sime, M.E., Green,T.R.G., & Guest, D.J. Scope marking in computer conditionals--a psychological evaluation. International Journal of Man-Machine Studies 9(1977),pp. 107-118.

SIME77b Sime, M.E., Arblaster, A.T., and Green, T.R.G. Reducing programming errors in nested conditionals by prescribing a writing procedure. International Journal of Man-Machine Studies 9(1977),pp-119-126.

SIME77c Sime, M.E., Arblaster, A.T., and Green, T.R.G. Structuring the programmer's task. Journal of Occupational Psychology 50(1977),pp.205-216.

WEIN74 Weinberg, G.M. and Schulman, E.L. Goals and performance in computer programming. Human Factors 16,1(1974),pp. 70-77.