In A.F. Blackwell & E. Bilotta (Eds). Proc. PPIG 12 Pages 181-192

The Effect of Programming Language on Error Rates of Novice Programmers

Linda McIver

School of Computer Science and Software Engineering
Monash University
linda.mciver@csse.monash.edu.au

Keywords: POP-I.B. barriers to programming, POP-II.A. novice programmers

Abstract

This paper describes the design and testing of a new introductory programming language, GRAIL1. GRAIL was designed to minimise student syntax errors, and hence allow the study of the impact of syntax errors on learning to program. An experiment was conducted using students learning programming for the first time. The students were split into two groups, one group learning LOGO and the other GRAIL. The resulting code was then analysed for syntax and logic errors. The groups using LOGO made more errors than the groups using GRAIL, which shows that choice of programming language can have a substantial impact on error rates of novice programmers.

Introduction

An experiment was conducted using a new language, GRAIL, which has been designed to minimise unnecessary errors. This experiment is the first in a series, in which GRAIL is being compared with other languages, some designed for introductory use and some used in industry. The aim of the experiments is to determine whether choice of language has any impact on error rates of novice programmers. The first experiment compares GRAIL with LOGO. While LOGO was not expressly designed for teaching programming as such, it was designed for children with no computing experience (Papert 1980).

This paper looks at the language design process, focussing on the design principles and the reasons for them, before examining the language comparison methodology and results. For a more detailed examination of the GRAIL language, see McIver and Conway (1999).

Although various arguments have been put forth for the use of a particular language or particular paradigm at the introductory level (Kölling, Koch, and Rosenberg, 1995; Conway, 1993; Meertens, 1981), there have been few, if any, empirical tests comparing different (entire) languages. The comparison of complete languages is difficult to do, particularly where the languages do not share the same paradigm. The tasks set may favour one language over the other, and the problem of what to measure is not easily solved. How does one measure programming knowledge in a language independent fashion?

One approach to language comparison is to record the way students interact with the language and environment – what kinds of mistakes they make, etc. - rather than attempting to measure language independent knowledge at the end of the course.

Trivial syntax errors are frequently ignored in the literature (Spohrer and Soloway 1986, Spohrer, Soloway, and Pope, 1985), perhaps because they are easily detected by the compiler/interpreter. Van Someren (1990) goes so far as to suggest that *"The syntax of most programming languages (including Prolog) is very limited and, in the initial stages only, a source of errors."*, although many

[1] "Genuinely Readable and Intuitive Language"

programmers will testify that the syntax of a language can continue to be a source of errors even for experts, as in the case of "==" in C, which even expert C programmers occasionally write as "=".

Although they may be easily corrected, these errors can still be disruptive and frustrating. Miller's (1956) discussion of the capacity of short term memory (7 plus or minus 2 items) suggests that experts have an advantage over novices, in that they make use of "chunking", where a single item in short term memory may be, in effect, an entire algorithm. In contrast, novices are forced to deal with details of syntax and individual elements of their algorithms, sometimes to the exclusion of the broader picture of the problem at hand. The more details a language requires the novice to recall, the less space is available for algorithms and the problem itself.

Language Design

Design Principles and rationales

There is one overriding principle on which GRAIL is based, namely that the language should facilitate learning to program. It is not sufficient for the language to be easy to program in – if that were the objective, a graphical interface would be sufficient to allow objects to be manipulated directly, or provide skeleton code that can be filled in by the user (for example, the structure of an **`if`** statement with a gap for the expression and the body of the **`then`** and **`else`** clauses).

As a consequence of the main design rule, two other design principles were specified as high level rules. These two rules are: "Maximize Readability" and "Minimize Unproductive Errors". Readability is largely self-explanatory, the main impact of this rule is the avoidance of terse and obscure operators that might benefit expert programmers but which may be confusing and hard to read for novices.

Unproductive errors can be defined as errors which impede and frustrate the novice, without providing good learning opportunities. Algorithmic errors from which novices can benefit can clearly not be done away with altogether, and they are an important part of the learning process. Trivial syntax errors, however, do not have the same obvious value, and may in fact impede learning as they distract students from the fundamentals of programming and problem solving.

The list of specific design rules which was applied to every feature considered for inclusion is somewhat longer. While a list of 7 rules was established at the beginning of the project, further rules became necessary during the design phase, as it became clear that some situations were inadequately covered. A list of the original 7 rules can be found in McIver and Conway (1996) and will not be covered in great detail here. These rules are:

- **Start where the novice is.** Make sure the language does not conflict with what the novice programmer already knows.
- **Use differing syntax to differentiate semantics.** Semantically different concepts should possess different syntax, to avoid confusion.
- **Make the syntax readable and consistent.** Readable and consistent to *novice* programmers.
- **Provide a small number of powerful, non-overlapping features.** More features require more syntax and greater complexity. The smaller the number of features included in the language, the simpler the language.
- **Be especially careful with I/O.** I/O can be surprisingly complex, as language designers endeavour to provide complete configurability. I/O should be simple and straightforward, providing sensible default behaviours.

- **Provide better error diagnosis.** Error diagnosis is a crucial aspect of learning to program, and compiler/interpreter error messages are the main form of interaction between student and machine.
- **Choose the appropriate level of abstraction.** The language should approximate the abstraction level of the problem domains in which novices find themselves.

The necessity for the following design rules became apparent during the design phase, as features were considered for inclusion in the language.

- **Avoid jargon.** Jargon is difficult to avoid, in part because it is difficult for an expert to identify jargon which will be unfamiliar to a beginner. Jargon in programming languages applies to both keywords and operators, as well as to the environment and support tools such as help systems and text books. Avoiding jargon involves avoiding using operators and keywords simply because they are programming "standards", or because they are familiar to the language designer, and instead choosing the most appropriate words and symbols for the construct. This is particularly challenging as it requires designers to identify their own cultural biases.
- **Favour simplicity over power.** Never include a feature simply because it is powerful - it must also be intuitive, readable and easy to understand and use. Powerful features are useful for experienced programmers, making complex programs simpler and easier to write, but for novice programmers an excess of powerful features can be counter-productive. ABC (Geurts, Meertens, and Pemberton, 1990), for example, has a list type which is automatically sorted. It is undeniably useful to be able to sort lists using built-in sorting functions, but to make sorting automatic can complicate the problem and confuse the student (McIver and Conway, 1996).
- **Avoid unexpected results.** Automatic and default behaviours can be extremely confusing for a novice. Implicit behaviours which silently change the state of a program or variable can be responsible for the violation of reasonable expectations. The behaviour of a construct should be able to be anticipated or assumed. Where behaviour is more complex than the syntax expresses, the complexity should be made explicit, with extra syntax if necessary.
- **Never complicate the simplest programs.** Any construct which introduces changes to the most basic operations, such as I/O or simple assignment and arithmetic, should be approached with extreme caution. The priority for a pedagogical language aimed at under-confident beginners is an easy, painless and confidence-building experience, particularly initially. The introduction of a complex construct which requires complication of simpler constructs for consistency should therefor be avoided. For example, the addition of file I/O should only be contemplated if it does not add complexity to the syntax for I/O using the screen and keyboard. eg. **`read length`** versus **`read length from keyboard`**, or **`read length from file`**.
- **Maximise "knowledge in the world".** Norman (1993) describes "knowledge in the world" as knowledge which need not be remembered, as it can easily be extracted from the world around us. In a programming language, knowledge in the world can be defined as syntax which clearly indicates the nature of the construct it represents. Maximum knowledge in the world means that the amount of details a programmer must remember can be minimised. In other words, each construct should be carefully crafted to contain as much information as possible in its keywords and general structure.

Principle features

Over the course of the design phase it became clear that, as with most user interfaces, the needs of experts and the needs of beginners are substantially different. In an attempt to aim the language squarely at the needs of absolute beginners, GRAIL has been deliberately designed to be an

extremely short term language. As such, many powerful concepts have been left out of the language, in order to make it as simple as possible.

The imperative paradigm was chosen for GRAIL, after much deliberation, because it represents a style of algorithmic expression that students are already familiar with (from previous interactions with instruction books, recipes, navigational directions, shampoo bottles, etc.), and because it remains the paradigm most widely favoured in actual teaching (imperative 48.7%, object-oriented and object-based 36.2%, functional (Scheme) 12%, functional (ML etc) 2.4%, other 0.7%; based on the Reid Report , Levy, 1995).

GRAIL does not contain pointers or references. While it is true that this restricts the range of problems which may be solved with the language, (for example a simple "swap" procedure is not possible) there is still a wide range of exercises which may be set, and the gain in simplicity is significant.

The use of the 6 Unicode characters, ≤ ≥ ≠ ÷ × ←, in GRAIL is intended to make the language agree as closely as possible with students' prior knowledge, particularly mathematical knowledge. The final character, ←, is used for assignment, in the hope of avoiding the confusion that '=' and its variants (such as `:=`) often provoke (McIver and Conway, 1996).

Strings in GRAIL are line-based, and the language possesses no string manipulation facilities. Like the exclusion of pointers and references, this allows the language to be considerably simpler (no need for a string to be an array at the same time), at the cost of a limit on the number and variety of programs which may be written in GRAIL. This also simplifies the provision of idempotent[2] I/O.

GRAIL contains a single arbitrary precision numeric type, eliminating the float/integer distinction still present in most languages. In addition, the language possesses static arrays, a simple structure type, and a small number of simple control structures.

Experiment

Subjects

The students who elected to take part in the study were first year university students about to start a course in computer science. None of the students had programmed before. Twenty-six students took part, divided into two groups, one learning LOGO and the other learning GRAIL. The same exercises were performed by each group, except for some minor variations initially, when the LOGO group experimented with turtle graphics.

Why LOGO?

LOGO was chosen as the language for comparison for the first study because it was designed for children, and as such presumably requires little or no prior computing knowledge. More languages will be compared in the future. In particular, real world languages such as C++, and languages more like GRAIL, such as Pascal, could produce interesting results.

Design of exercises

Exercises for the study were designed to be simple problems which require fairly short programming solutions, but which also encourage the use of various language constructs, such as structures, arrays, loops and functions. Comparing two such different languages made the appropriate choice of language-independent exercises difficult, and this may have affected the outcome in unexpected ways. Smaller problems were easily matched – a function to calculate the square of a number is equally trivial in each language – however larger problems may have unintentionally favoured one

[2] Idempotency of I/O implies that the input and output format for any type are exactly equivalent. That is, if the output of one variable is used as the input to another, the effect is exactly equivalent to assignment between the two.

language or the other. A balance was struck between using the same exercises in each language, and still allowing language specific experimentation, in particular the use of the LOGO turtle was encouraged in the early stages of the course, in order to help students become confident and familiar with the environment.

The 8 exercises set included trivial programs such as reading in a name and printing out "hello *name*", calculation of compound interest, and more complex programs such as creating a list of students names, id numbers and marks, and identifying the top student in a class.

Interface and Development Environment

A single interface was developed in order to make the interface and development environments of the two languages as similar as possible, and to remove them as a source of variation in the results. The development environment consisted of a simple text editor with basic edit functions (cut, copy, paste, undo/redo) and file functions (new, save, load, quit). The editor also possessed a large button marked "RUN" in the bottom right hand corner. No distinction was drawn between compiling and running a program. Clicking on the RUN button compiled a GRAIL program and interpreted a LOGO program. An error caused a small bug to appear next to the relevant line of code. Clicking on the bug displayed the error message.

Data collected

During the course, each time the RUN button was pressed, a copy of the code was collected. Duplicate programs were removed (where the same program was compiled twice), and errors which persisted through more than one compilation were only counted once.

Data Analysis and Results

Categorizing errors

Two broad categories were used in the analysis of the code - syntax and logic errors. Errors classified as syntax errors were generally simple errors that did not indicate a serious misunderstanding, an incorrect algorithm, or an incorrect translation from algorithm to code. These errors included typographical errors, use of incorrect keywords (eg "**`end`**" instead of "**`exit`**" in GRAIL, "**`loop`**" instead of "**`repeat`**" in LOGO); incorrect punctuation (eg '**`"`**' instead of '**`:`**' in LOGO, or a missing "**`,`**" in a "**`write`**" statement in GRAIL); missing keywords (eg missing "**`end`**" statements in both languages); incorrect operators; missing quotes or brackets.

Errors classified as logic errors generally involved a flawed algorithm. These include failure to increment a loop counter; accessing an array position that doesn't exist (ie violating the bounds of an array); prompting for the user to enter a value after the value has been read eg

```
read myNumber
write "please enter my number"
```

using the wrong variable; and incrementing a loop counter in the wrong place (ie outside the loop, or before the loop counter is used for the first time).

These categories are broad and imprecise, but they do make a distinction between algorithmic errors and those which clearly arise from the features of the language.

Results

Figures 1 and 2 show that the error frequencies for the GRAIL groups were substantially lower than those for the LOGO groups. The mean number of syntax errors over the course of the programme for the LOGO group was 31.08, with a standard deviation of 8.33, versus a mean of 13.62 and standard deviation of 7.98 for GRAIL. Logic errors gave a less dramatic result, but still significant, with

LOGO students making 17.77 errors on average (standard deviation 7.05), versus 9.54 (standard deviation 5.39) for GRAIL.

An independent t-test shows that the two groups are significantly different for both syntax errors ($t = 5.24$, $df = 24$, $p < 0.01$) and logic errors ($t = 3.21$, $df = 24$, $p < 0.01$). These results support the hypothesis that the choice of language, or at least the type of language chosen, does make a difference to the type and frequency of errors made by novice programmers.

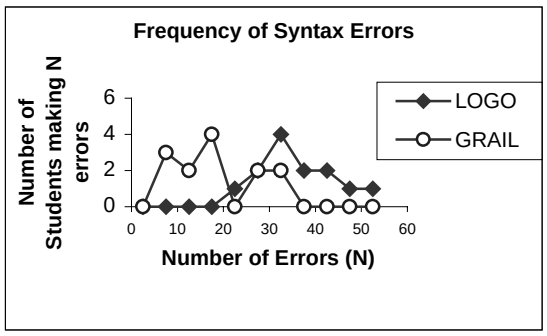

Figure 1: Distribution of Syntax Errors

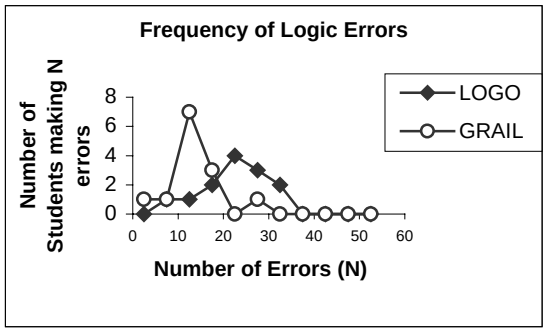

Figure 2: Distribution of Logic Errors

Correlation between syntax error rates and logic error rates

The mean error rates for logic and syntax errors in the two groups suggest a link between the number of syntax errors made and the likely number of logic errors. However, the correlation coefficient is not high ($R=0.56$, $df=24$, $p<0.01$), and Figure 3 shows that the relationship between syntax and logic errors is not clear – students with a high number of syntax errors did not always exhibit a correspondingly high number of logic errors.

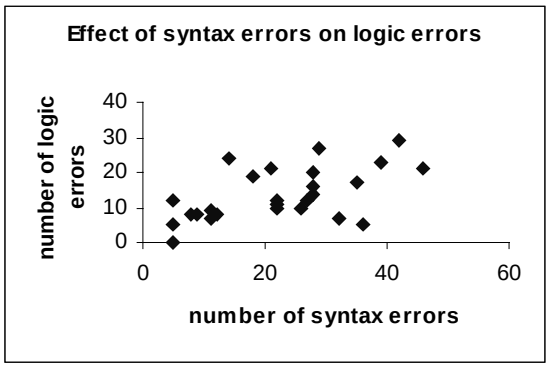

Figure 3: Correlation between syntax and logic errors

Other factors may be contributing in unexpected ways. For example, the amount of code written varied between students, as did the success rate on individual exercises. Further investigation with greater numbers of students will shed more light on this problem.

Anecdotal evidence

Three students in the GRAIL group went further than the course syllabus, either solving problems of their own devising or extending the problems given during the course. No students in the LOGO group experimented to this degree. Given the low number of students in total, this is an interesting, but not significant result. It is possible that the language difference somehow encouraged greater experimentation, or it could be coincidence that the three more adventurous students all used GRAIL. These three students may also have encouraged one another. Once again more testing is required to determine whether there is a significant difference.

Conclusions

Despite the small number of students in the trial, the substantial difference in error rates between the two languages merits further investigation. The results support the hypothesis that syntax errors, even easily corrected ones, have a substantial impact on the experience of learning to program. As well as further experiments using a wide variety of languages, tests of learning outcomes, perhaps between similar languages such as GRAIL and Pascal, would be useful in determining the connection, if any, between error rates and learning outcomes.

Acknowledgements

Thank you to Alan Blackwell for encouraging me in the direction of PPIG, to Paul Brna for advice on PPIG papers, and to Damian Conway and Tony Jansen for proof reading and sanity checking.

References

Conway, D. (1993) Criteria and Consideration in the Selection of a First Programming Language, *Technical Report 93/192, Department of Computer Science, Monash University.*

Dalbey, J. and Linn, M. (1985) The demands and requirements of computer programming: a literature review. *Journal of Educational Computing Research*, 1(3), 253-274.

Geurts, L., Meertens, L. and Pemberton, S., (1990) *ABC Programmer's Handbook*, Prentice Hall.

Kölling, M., Koch, B., and Rosenberg, J. (1995) Requirements for a First Year Object Oriented Teaching Language, *SIGCSE Bulletin*, 27(1), 173-177.

Levy, S. (1995) Computer Language Usage in CS1: Survey Results, *SIGCSE Bulletin*, 27(3), September, 21-26.

McIver, L. and Conway, D. (1996) Seven Deadly Sins of Introductory Programming Language Design. *Proceedings, Software Engineering: Education & Practice 1993 (SE:E&P'96),*309-316.

McIver, L. and Conway, D. (1999) GRAIL: A Zeroth programming language. *Proceedings, International Conference on Computing in Education (ICCE99)*, 43-50.

Meertens, L. (1981) Issues in the Design of a Beginners' Programming Language, *Algorithmic Languages*, de Bakker/van Vliet (eds), IFIP North Holland Publishing Company, 167-184.

Norman, D. (1993) *Things that make us smart: Defending human attributes in the age of the machine*. Addison-Wesley.

Papert, S. (1980) *Mindstorms: Children, Computers, and Powerful Ideas.* The Harvester Press Limited.

Spohrer J. C. and Soloway, E. (1986) Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*. 29(7), 624 - 632 .

Spohrer, J. C., Soloway, E. and Pope, E. (1985), Where the bugs are. *Proceedings CHI '85*. 47-53.

Van Someren, M. W. (1990) What's wrong? Understanding beginners' problems with Prolog. *Instructional Science.* 19. 257-282.