An Analysis of Programming Aptitude of Engineering Undergraduates

Harley R. Myler

Abstract

This paper presents the results of an evaluation of programming aptitude of incoming engineering undergraduates at the onset and terminus of a basic engineering programming course. A brief introduction covering the fundamentals of algorithms and the basic architecture of computers was given to a class of mixed-discipline engineering undergraduates and then followed with an examination to test the ability of the students to form an algorithm to solve a problem. The algorithm was constrained to a limited set of instructions that were intended for a person to execute manually. The test was designed to include variable/register assignment, transfer and arithmetic operations, conditionals, loops, and I/O. At the end of the course, a similar problem requiring use of a high-level language, in this instance C, was used to gauge the effectiveness of instruction.

Introduction

The teaching of computer programming to engineering students can be both rewarding and frustrating due to the nature of the material. Many students struggle with the abstractions of programming and at some point during the course experience a "Eureka!" effect as the concept of telling a machine what to do suddenly becomes clear. The student is then off and running with their newly discovered ability. This discovery has been observed to occur at different times during the semester that the course is offered, although there are some students that seem to understand programming from the beginning, others may not achieve enlightenment until almost the completion of the course. What is ubiquitous, however, is the fact that the understanding appears to gel and the entire set of material presented is clarified at once. This effect is easily observed, and to some extent measured, by the level of programming capability displayed.

A number of questions arise based on the phenomenon described. Are specific key concepts particularly difficult such that a specialized approach is needed to explain them? Is a refractory period, known to research as the psychological refractory period (PRP), needed for programming concepts to become clear? Does the PRP vary by individual? Formally, the PRP is defined as a slowing in the reaction times to stimuli in one task when another task is performed concurrently, but it has also been used to describe the period of delay that occurs when tasks or concepts are learned. Since the time to understanding appears to vary between individuals, this opens the possibility that the acquisition of programming ability may be influenced by nature or nurture; hence, it will be useful to explore the status of what is considered to be an ongoing debate.

1 School of Electrical Engineering & Computer Science,
University of Central Florida; Orlando, Florida 32816-2450.  hrm@engr.ucf.edu
Nurture vs. Nature

Across a variety of disciplines that include anthropology, psychology and others, there is a great deal of argument and study over the origins of human behavior. Are some human behaviors innate? Do we learn everything from our parents and/or surroundings? Are some behaviors too complex to be explained by one source? Researchers in many fields have taken different approaches to these questions. Some attempt to answer these questions by looking at the specific anatomy involved, while others look to primates to understand our own behaviors. Adam Kuper’s book, The Chosen Primate, discusses many behaviors, such as language and intelligence, which may be related to the study of programming languages.

Language is one of the fundamental components of human society and because of this, language continues to be a large area of study. Darwin suggested that there is a feedback loop between culture and language in that the development of the brain allowed for the creation of language, which stimulated even further brain development. From this view one may conclude that language is both culturally determined and biologically imprinted. Although many animals communicate with each other, researchers cannot look to primate studies for information about language simply because human language is unique.

Human intelligence has always been a source of intense debate, especially within the nature vs. nurture argument. Nevertheless, researchers have come further in the field since the days of Phrenology, where early psychologists related topographical features of the head to cognition. Francis Galton, the founder of Eugenics, was very interested in finding out where intelligence came from. This prompted his belief that, "the competition between the races was won by the population that produced the most men of genius." Suggesting that the white race was in control because of better genes. Similarly, Lewis Terman, the psychologist who wrote the Stanford-Binet intelligence test, was convinced that, "the children of successful and cultured parents test higher than children from wretched and ignorant homes for the simple reason that their heredity is better." These conclusions were grossly simplified. How does one explain geniuses borne to ordinary people? What role does education and opportunity play in determining the source of intelligence? Can we overcome our genes through our culture? Perhaps genetics only provides a foundation for intelligence, which can be pushed even further by learning.

Clearly many questions remain to be answered in the areas of language and intelligence. Contemporary research in these areas has produced evidence that continues to show that there is combination of cultural and biological determinants in these cases and it is clear that these behaviors will most likely be studied for a long time to come. Notwithstanding the fact that human behaviorists have concluded that nature and nurture are not mutually exclusive, how can this fact be applied to the teaching of computer programming? One approach is to try to identify and isolate a "conceptual hurdle" and use that concept to improve instruction.

Repetitive Control Structures

One of the more difficult concepts for students to grasp seems to be that of repetitive control structures, or loops. This is somewhat understandable since perceptually the process is one of going backward. Students appear to grasp conditional structures readily, since these are ubiquitous in our daily thinking process. A popular ad campaign bemoans the fact that choices abound, "coffee or tea?", "car or train?", "boxers or briefs?", etc., etc. and that day to day living is complicated by this; however, these choices are foundational to our daily lives and a precursor to the more formal clause formed of predicate and consequent. When programming, the difficulty lies primarily in the syntax of the conditional and that is rapidly mastered. This opens the student to an extensive amount of programming power in the sense of programs that require decision-making as part of the algorithm.
With the addition of arithmetic and logical operators and simple I/O functions, the student can begin to compose programs that illustrate a powerful aspect of the digital computer. Simple programs such as an arithmetic tutoring program like that shown in Figure 1 can be assigned.

```c
#include <stdio.h>
void main(void){
    float answer;
    printf("What is 1 + 1?\n");
    scanf("%f", &answer);
    if(answer == 2) printf("Correct!\n");
    else printf("Incorrect, sorry the answer is 2.\n");
    
    
}
```

Figure 1. Math tutor program illustrating use of conditional.

The other powerful aspect of computing comes with the effective use of looping structures. For some reason, the programming of loops is difficult for some students. There may be a human factors element at work in that the loop causes the program to "go backward" and this is contrary to the way that humans read and process information. Most people would agree that a common aggravation in reading some magazine articles is the way that portions of the article may be distributed throughout the volume. When program loops are multidimensional, they pose an additional problem to the student who has not come to grips with how loops function. As a consequence, it was concluded that the concept of nested-loops would be a good test problem to try and illuminate the question of programming aptitude and the effectiveness of instruction.

**Testing**

In order to evaluate the concepts discussed earlier, a pretest was prepared that focussed on the ability of the student to grasp fundamental programming techniques. Specifically, the concepts of: algorithm, program, instructions and conditional and loop constructs. Since this exam was given as a test of comprehension of abstract concepts, it was necessary that the students first receive instruction on the definitions of the algorithm and program and how an algorithm is developed to solve a problem. Additionally, the students were given introductory instruction in computer development history, basic architecture concepts to include the von Neumann machine and the Tanenbaum Virtual Machine Hierarchy. The test, shown in Figure 2, requires the student to develop a set of instructions for a person such that the individual produces a simple multiplication table. The students were told that the person receiving the instructions is an *idiot savant* and could only understand the instructions listed in Figure 3 and no others. Clearly, the goal was to simulate a computer-programming problem *without* involving a computer explicitly and having to deal with the requisite restrictions of syntax and implementation. The test was administered to a class of 41 students and they were informed that the exam was not for grade. After the exam was collected, a solution to the exam, Figure 4, was reviewed and explained. At the end of the course, following a semester of C programming instruction, a post-test was administered. The post-test required that the student write down a C program to produce a conversion table of zero to two pi radians of degree, sine and cosine values.
Consider the following array of numbers:

```
  1  2  3  4  5  6  7  8  9 10
  2  4  6  8 10 12 14 16 18 20
  3  6  9 12 15 18 21 24 27 30
  4  8 12 16 20 24 28 32 36 40
  5 10 15 20 25 30 35 40 45 50
  6 12 18 24 30 36 42 48 54 60
  7 14 21 28 35 42 49 56 63 70
  8 16 24 32 40 48 56 64 72 80
  9 18 27 36 45 54 63 72 81 90
 10 20 30 40 50 60 70 80 90 100
 11 22 33 44 55 66 77 88 99 110
 12 24 36 48 60 72 84 96 108 120
```

Write a numbered set of detailed instructions to an individual such that they reproduce the array on a sheet of paper using a calculator, pencil and three erasable pads labeled X, Y and Z that can have a single number written on them at any time. This person only understands a limited set of instructions.

**Figure 2. Test program for programming aptitude.**

- write <some number> to pad <X, Y, or Z>.
- copy the number on pad <X, Y, or Z> to the paper.
- add 1 to the number on pad <X, Y, or Z>.
- calculate <X, Y, or Z> < > <X, Y, or Z>.
- write C, the calculated value to the paper or <X, Y, or Z>.
- compare the values on the X, Y, or Z pads to each other and go to a different step or stop based on that value, e.g.,
  - if Y is greater than or equal to X, then go to step 2
  - start a new line (row of numbers).

**Figure 3. Allowable instructions for test program.**

1. Write 12 to Z
2. Write 1 to Y
3. Write 1 to X
4. Calculate X x Y
5. Write C to the paper
6. Add 1 to X
7. If X is less than or equal to Z,
   then go to step 4
8. Start a new line
9. Add 1 to Y
10. If Y is less than or equal to Z,
    then go to step 3

**Figure 4. Solution to test program.**
Results and Conclusions

The exams were evaluated with a six-part scheme, listed in Figure 5, which focussed on the ability of the students to develop the proper logic and a realistic algorithm to solve the problem. Of the 41 students who sat for the pre-test, only 33 took the post-test. The results, shown graphically in Figures 6 through 12, are comparative between the pre- (red) and post- (blue) tests and the vertical axis is percentage of 33 test takers. On the whole, the results were surprisingly good and indicated that in the majority of cases the students did better after the semester of instruction. This trend was not true, however, for the number of syntax errors (Figure 9). Here the pre-test results were better in terms of no errors, but not nearly as good in terms of excessive errors. This is easily explained since the pre-test was written using a simplistic, toy language whereas the post-test was written in C. Figure 12, the number of statements used, was a criteria to determine which students attempted the trivial solution to the problem, i.e., using multiple output statements to produce the tables without a loop. This was more apropos to the pre-test where 3 students chose to not use a loop mechanism.

This approach was experimental and needs more work on the structure and content of the pre- and post- tests as well as a less subjective approach to the evaluation of the exams. Notwithstanding this deficiency, the study produced an interesting and valuable insight into the programming instruction dilemma.

<table>
<thead>
<tr>
<th>Logic</th>
<th>Bad Instructions (instructions not in the instruction set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -- logic correct; algorithm had two loops, one each for row and column generation</td>
<td>B0 -- no bad instructions</td>
</tr>
<tr>
<td>2 -- logic almost correct; algorithm only had one loop</td>
<td>B1 -- two or fewer bad instructions</td>
</tr>
<tr>
<td>3 -- logic incorrect, no loop mechanism at all (no if statement)</td>
<td>B2 -- less than 5 bad instructions</td>
</tr>
<tr>
<td>LT -- algorithm had a loop termination mechanism; i.e., a variable was identified (one of the pads) that was set to test whether the row or column was complete.</td>
<td>B3 -- excessive bad instructions</td>
</tr>
<tr>
<td>NA -- no algorithm or meaningless algorithm; student did not write down an algorithm or the algorithm was so primitive as to be meaningless.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax (badly formed or worded instructions)</th>
<th>Program Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -- no syntax errors</td>
<td>P1 -- program works</td>
</tr>
<tr>
<td>1 -- two or fewer syntax errors</td>
<td>P2 -- program partially works</td>
</tr>
<tr>
<td>2 -- less than 5 syntax errors</td>
<td>P3 -- program would not work</td>
</tr>
<tr>
<td>3 -- excessive syntax errors</td>
<td></td>
</tr>
</tbody>
</table>

Program Statements

| I0 -- program had 12 or fewer statements | I1 -- program had greater than 12 but less than 50 statements |
| I2 -- program had greater than 50 statements | |

Figure 5. Evaluation scheme.
Figure 6. Program Logic

Figure 7. Loop Terminator

Figure 8. Algorithm
Figure 9. Syntax Errors

Figure 10. Bad Instructions

Figure 11. Program Operation

Figure 12. Statements
References


Harley R. Myler

Harley R. Myler is a Professor of Electrical Engineering in the School of Electrical and Computer Science at the University of Central Florida (UCF) in Orlando and holds a joint appointment in Computer Engineering. Dr. Myler graduated from Virginia Military Institute in 1975 with the BSEE degree and did his graduate work at the Electronic Vision Analysis Laboratory at the New Mexico State University earning the MSEE in 1981 and the Ph.D. in 1985. He is currently the Director of the Machine Intelligence and Imaging Laboratory at UCF and has published over 60 articles and four books in the areas of imaging science and engineering, signal processing and engineering education. Dr. Myler is a registered professional engineer in Florida and is a consultant to defense contractors and systems companies. He is a senior member of the IEEE, a member of the SPIE, a Tau Beta Pi eminent engineer and a member of Eta Kappa Nu.