Early Relational Reasoning and the Novice Programmer: Swapping as the "*Hello World*" of Relational Reasoning

Malcolm Corney
Faculty of Science and Technology
Queensland University of Technology,
Brisbane, QLD, Australia
m.corney@qut.edu.au

Raymond Lister
Faculty of Engineering and Information Technology,
University of Technology, Sydney,
Sydney, NSW, Australia
Raymond.Lister@uts.edu.au

Donna Teague
Faculty of Science and Technology
Queensland University of Technology,
Brisbane, QLD, Australia
d.teague@qut.edu.au

Abstract

We report on a longitudinal research study of the development of novice programmers in their first semester of programming. In the third week, almost half of our sample of students could not answer an explain-in-plain-English question, for code consisting of just three assignment statements, which swapped the values in two variables. We regard code that swaps the values of two variables as the simplest case of where a programming student can manifest a SOLO relational response. Our results demonstrate that the problems many students face with understanding code can begin very early, on relatively trivial code. However, using traditional programming exercises, these problems often go undetected until late in the semester. New approaches are required to detect and fix these problems earlier.

Keywords: Novice programmer, SOLO, chunking.

1 Introduction

Over the last six years, the BRACElet project has studied the relationship between the ability of novice programmers to write code and explain code. Two of the earliest BRACElet papers (Whalley *et al*., 2006; Lister *et al*., 2006) studied how students answered the following explain-in-plain-English question in an end-of-first-semester programming exam:

> In plain English, explain what the following segment of Java code does:

```
bool bValid = true;

for (int i = 0 ; i < iMAX-1 ; i++)
{
   if (iNumbers[i] > iNumbers[i+1])
          bValid = false;
}
```

The BRACElet researchers analysed student responses to this question in terms of the SOLO taxonomy (Biggs and Collis, 1982). Some students of course provided responses that were inadequate, vague or simply incorrect, but there were also correct and comprehensive responses from students that fell into two SOLO categories:

- **Multistructural:** This is a response in which the student provides a description of what each line of the code does, without linking the lines together.
- **Relational:** This is a response in which the student provides a correct summary of the overall computation performed by the entire piece of code, such as, for the above code "*it checks to see if the array is sorted*". We refer to the ability to read a piece of code and deduce the overall computation performed by that code as *relational reasoning*.

Since those first two BRACElet papers, replication studies have tried several variations on the format of explain-in-plain-English questions. Lister and Edwards (2010) provided a summary of those variations. From all those studies, it appears that there are some students who are able to provide multistructural responses, but who struggle to perform relational reasoning.

In another BRACElet study, Lopez et al. (2008) linked relational reasoning with code writing. They found that a combination of student scores on tracing tasks and the ability to manifest relational reasoning on explain-in-plain-English questions accounted for 46% of the variance on a code writing task. Replication studies have reported similar results (Lister, Fidge and Teague, 2009; Venables, Tan and Lister, 2009; and Lister et al., 2010).

All of the above BRACElet studies used data collected as part of end-of-first-semester exams. Also, most of those studies used explain-in-plain-English questions where the code involved iterative processes on arrays.

1.1 Relational Reasoning without Iteration

The motivation for our study came from a colleague, who had taught first-semester programming classes for many years, and who had won teaching awards while doing so. Our colleague made the assertion that the first few weeks of teaching programming are straightforward, but the problems start with the introduction of loops.

That comment by our colleague led us to wonder – does relational reasoning only become a problem for novices when loops are introduced? We then looked at examples of code that textbooks presented to students prior to the introduction of loops. (All of the code-writing problems we examined were in the procedural paradigm.) We found that one common type of example presented to

students could be characterized as *input-calculate-output*. For example, consider a piece of code that calculates the area of a rectangle, given the length and height. Such a piece of code has the following general form:

```
Input a value into a variable;
Input a value into a second variable;
Assign to a third variable the
     result of the calculation;
Output the third variable;
```

Such code has properties that make it easier to understand than the iterative code studied later in semester:

- All the variables are either directly manipulable by the user (i.e. input) or directly observable by the user (i.e. output). Thus all variables have a meaning defined by the "real world" problem to be solved, without reference to any algorithm.
- Given the "real world" definition of each variable, the purpose of each line of code makes sense in isolation from the other lines.
- None of the lines of code involve overwriting a meaningful variable value with a new value. Such a change to a variable would change the exact "real world" problem being solved.

Could it be, then, that the first few weeks of semester, prior to the introduction of loops, are straightforward because the non-iterative code we traditionally present to our students only requires a form of reasoning that is simpler than the relational reasoning required for understanding iterative code? If so, could we identify non-iterative pieces of code that might better prepare students for loops?

The above questions led us to identify the simplest piece of non-iterative code that requires the same form of relational reasoning as iterative code − the swapping of the values between two variables:

```
temp = a;
   a = b;
   b = temp;
```

Unlike the code for calculating the area of a rectangle:

- the variable `temp` is not an input/output variable but only has meaning in the context of an algorithm;
- each line does not stand alone. For example, the final line does more than simply assign the value of `temp` into `b` − it assigns *the original value of* `a` into `b`; and
- the meaningful values in variables `a` and `b` are overwritten.

Since some programming textbooks use a "Hello World" program as their very first example, we refer to code that swaps the values of two variables as the "Hello World" of relational reasoning. We then asked the following research question, which we pursue in this paper:

> *Would some students struggle in the early part of the semester with the code for swapping the value of two variables, just as earlier BRACElet research had demonstrated that students struggled with iterative code at the end of semester?*

If there are students who struggle with the code for swapping two variables, then the early part of semester does not prepare students well for the iterative code to follow.

Prior to our study, there was an earlier BRACElet study that used an explain-in-plain-English question for swapping two variables (Sheard *et al*., 2008). However that study, like other BRACElet studies, examined students at the end of the first semester.

2 The Learning Context

The students from whom data was collected for this study were enrolled in an introductory computing course. This subject was a breadth-first introduction to building IT systems and was not devoted entirely to the teaching of programming. Other material taught during this subject included introductions to SQL and web-page development. The first six weeks of the 13 week semester were allocated to an introduction to programming, using Python. In those six weeks, the students were expected to reach a point where they could understand and write code involving array/list structures, conditional statements, loops, function definition and use, and recursion**.** In the remaining seven weeks of the semester, students had further practice with their Python skills, when they used Python in the web-based systems they wrote (e.g. to interact with SQL databases and to perform input/output processing for web pages).

This paper is restricted to an analysis of the performance of the students on the programming component of this subject. For more details on the entire subject, see Corney, Teague and Thomas (2010).

3 Method

Students attending the lectures in weeks 3 and 5 were given two short written tests. These two tests are provided in this paper, as the last two pages.

Normal exam conditions applied during these tests. The lecture room was supervised by the first and third authors to ensure that students completed the exams individually. There was no strict time limit on either test. Students were given as long as they needed. After 10 to 15 minutes all students had finished. Most had completed the test well before that time.

Prior to both tests, the students were told that the tests would not contribute to their grade. The students were told that the teachers would use the test results as a guide to what topics required more teaching or improved methods of teaching. Of the test sheets returned by the students for marking, a very small number were entirely blank, and a few were completed but left anonymous. The anonymous tests were excluded from the analysis described below, since we could not collate a student's performance across tests without knowing their identity.

Following each test, the first author, who taught the subject, talked about each of the questions. He demonstrated an approach to solving the questions, and he provided correct answers to the questions.

3.1 The Week 3 Test

This first test was administered in week 3 of the semester, at which time the material being presented in lectures assumed that students could understand the basic

concepts of variables and assignment statements. The test was distributed to students on both sides of a single sheet of paper. We provide this test on the second last page of this research paper, reduced in size to conserve space. A total of 227 students submitted this test.

3.1.1 Questions 1 to 3: Screening Questions

The three goals of these screening questions was to test that a student (1) understood variables, (2) understood assignment, and (3) could trace code of similar complexity to the remaining questions in that test. Since this test was administered very early in the semester, we could not assume that students had these skills, particularly as some students enrolled late.

In this paper, we are not interested in what percentage of the class understood variables and assignment statements. Asking these three screening questions is analogous to, in a non-programming research study, first giving an experimental subject a test on their ability to read English. It is sometimes a wise precaution to know that someone can read English before giving that person a test on the real material of interest, when the test on the real material happens to be written in English – it is the experimental subject's grasp of the real material that we would want to measure, not the experimental subject's ability to read English. More specifically, our research interest in this paper involves testing whether a novice programmer understands a piece of code as a whole, when the novice understands all the programming constructs in that code. We therefore need to screen to ensure that students understand those constructs. Students who could not successfully answer all three of these questions were eliminated from the analysis we present later in the "Results" section.

A fourth goal in having the screening questions was to ensure that we had a sample of students who had made a genuine effort to answer the questions in the test. Students who scored a perfect score on these screening questions clearly approached the test seriously.

3.1.2 Questions 4 and 5: Explain a Swap

Both question 4 and question 5 are explain-in-plain-English questions, and the code in both questions swaps the values of two variables, using a third variable as a temporary variable.

At week 3 of the semester, students had not encountered an explain-in-plain-English question before, so there was a danger that the students might not understand the type of answer we wanted. For that reason, we designed Question 4 so that it would show the students what sort of answer we wanted, in three ways:

- Question 4 begins by providing an example of the type of answer we wanted, "*The purpose of the following three lines of code is to swap the values in variables a and b*".
- Question 4 specifies that the answer should be "*one sentence*". Furthermore, the box in which the students are directed to write their answer is meant to indicate that the answer should not be very long.
- Question 4 contains the note "*Tell us what the second set of three lines of code do all by themselves. Do NOT think of those second three lines as being executed after the first three lines of code.*" We added this note after a pilot test, at a different university. In the pilot, we found that a small number of students gave answers such as "*It restores the variables to their original values*" because the students thought of the second set of three lines as being executed after the first set of three lines.

The aim of Question 5 was to see whether students could generalise from Question 4, and see that Question 5 also swapped values between two variables. We can report that 91% of the 227 students who answered both Question 4 and Question 5 were consistent across those questions – either they answered both of these questions correctly and relationally, or they answered both questions either incorrectly or non-relationally. The remaining 9% were split roughly even, among some who answered Question 4 correctly and relationally, and some who answered Question 5 correctly and relationally. Given that 91% of students answered both questions consistently, we subsequently focussed our analysis on Question 4 and ignored Question 5.

3.2 The Week 5 Test

This second test was administered in week 5 of the semester, at which time the material being presented in lectures assumed (in addition to the concepts tested in week 3) that students could understand `if` statements. Like the week 3 test, this test was distributed to students on both sides of a single sheet of paper. We provide this test on the last page of this research paper, reduced in size to conserve space. A total of 176 students submitted this test, of whom 148 had also completed the week 3 test.

3.2.1 Question 1: Write a Swap

The first question required students to write code that swaps the values between two variables. Recall that Question 4 in the week 3 test had asked students to explain a piece of code that also swapped two variables. The research interest in asking students to write the swap code was to see whether students, two weeks after the first test, could remember the swap code well enough to write it.

3.2.1.1 Two Temporary Variables

Of the 140 students who answered this question, 8 students (6%) made appropriate use of two temporary variables, instead of the minimum necessary single temporary variable. (Recall that the swap code in the week 3 test used a single temporary variable, so these students had clearly not memorised that code.) When a student's code with two temporary variables worked correctly, it was marked as correct, since such an answer was not excluded by the phrasing of the question. Another 9% of students also used two temporary variables, but they did so incorrectly.

3.2.2 Question 2: Screening Questions

The second question performed a similar screening role in the week 5 test as the first three questions of the week 3 test. That is, the goal of Question 2 was to test that a student could trace code containing `if` statements, which also implied that the student had made an effort on the test. As for the week 3 test, students who did not

successfully answer this question were eliminated from the analysis we present later in the "Results" section.

3.2.3 Question 3: Explain a sort of three variables

The third question is an explain-in-plain-English question, in which the code contains `if` statements. In the framing of this question, we took steps similar to those steps we took in the framing of Question 4 in the week 3 test, to ensure that students were clear on what type of answer was required.

3.3 End of Semester Exam

At the end of the 13 weeks of semester, the students underwent an examination of the material from the entire semester. In this research paper, we shall focus on three programming-related questions from that exam, which are described below.

3.3.1 MCQ: Explain Product of Even Numbers

Of the eight programming-related multiple choice questions in the final exam, only one was an explain-in-plain-English question:

> Which best describes the purpose of the following Python function definition?

```
def do_something_with_numbers():
    total = 1
    response = input('Please input an integer: ')
    while (response != 0):
        if response % 2 == 0:
            total = total * response
        response = input('Please input an integer: ')
    return total
```

(a) It does not do anything as the body of the while loop never executes
(b) It returns the product of all numbers entered
(c) It returns the product of all even numbers entered ← *the correct option*
(d) It returns the product of all odd numbers entered

3.3.2 Trace a Swap

One of the programming-related exam questions tested the students again on code that swapped the values in two variables:

> What do the variables value_1, value_2 and value_3 hold after the following Python code is executed? Assume that they are all integer type variables.

```
value_1 = 10
value_2 = 15
value_3 = value_1
value_1 = value_2
value_2 = value_3
```

Students were deemed to have supplied a correct answer if they provided the correct values for all three variables.

3.3.3 Write the Reverse of a String

Only one exam question required students to write any Python code:

> The following Python source code copies a String:

```
source = 'the cat sat on the mat'
target = ''
for character in source:
    target = target + character
```

> Rewrite this code snippet so that the target String contains the source String in reverse order. e.g. 'abc' becomes 'cba'.

A concise, correct answer needed only to be a copy of the above code, with the final line changed to:

```
target = character + target
```

Some students provided a more verbose but correct answer, such as the following:

```
n = 0
target = ''
for character in source:
    target = target + source[len(source) - n - 1]
    n += 1
```

In this paper, we are interested in the students' conceptual grasp of programming, and not their ability to get code exactly right, first time, under exam conditions. Therefore, we ignored minor bugs. For example, some students provided a similar solution to the more verbose solution provided above, but they made errors in the calculation of the subscript into the sequence `source`. For example, instead of the correct `(len(source) - n - 1)` as in the above solution, some students wrote `(len(source) - n)`. We ignored such errors. We also ignored off-by-one errors in loops.

The most common answer attracting zero marks used code similar to the concise answer given above, but replaced the plus sign in `target = target + character` with a minus sign: `target = target - character`. Since the subtraction operator does not exist for strings, those students were either manifesting a conceptual error, or were making a guess.

4 Results

As described earlier, we culled all students who did not correctly answer all four screening questions (i.e. the first three questions in the week 3 test, and the second question from the week 5 test). We also culled all students who had not provided some form of answer to all of the remaining questions, except Question 5 from the week 3 test, since that question was left out of our data analysis. (The reasons for leaving it out were discussed in section 3.1.2). After this culling, 83 students remained. The percentage of these students who answered each test and exam question correctly is shown in Table 1.

Week 3	Week 5		End of Semester Exam		
Explain a swap	write a swap	explain a sort of 3 variables	MCQ, explain the product of even nums	trace a swap	write the reverse of a string
47%	73%	48%	76%	89%	59%

Table 1: The percentage of students who answered each question correctly (n=83)

Week 3 explain a swap	Week 5		End of Semester Exam		
	Column A	Column B	Column C	Column D	Column E
	write a swap	explain a sort of three variables	MCQ, explain the product of even nums	trace a swap	write the reverse of a string
Wrong (n = 44)	57%	36%	64%	82%	41%
Right (n = 39)	92%	62%	90%	97%	79%
$\chi 2$ test	p = 0.001	p = 0.03	p = 0.01	p = 0.03	p = 0.001

Table 2: The performance of students, broken down according to the week 3 explanation question (n=83).

Week 5 write a swap	Week 5	End of Semester Exam		
	Column A	Column B	Column C	Column D
	explain a sort of three variables	MCQ, explain the product of even nums	trace a swap	write the reverse of a string
Wrong (n = 22)	14%	59%	68%	27%
Right (n = 61)	61%	82%	97%	70%
$\chi 2$ test	p = 0.001	p = 0.03	p = 0.001	p = 0.001

Table 3: The performance of students, broken down according to the week 5 writing question (n=83).

4.1 Results for Week 3 Explain a Swap

Table 2 shows the percentage of students who correctly answered questions from the week 5 test and the end of semester exam. These percentages are broken into two rows, according to how students answered the explanation of a swap in Question 4 of the Week 3 test. The row that commences with the word "Right" shows the percentages for the 39 students who correctly answered Week 3 Question 4, while the row commencing "Wrong" shows the percentages for the 44 students who answered incorrectly.

As the bottom row of Table 2 shows, chi square analysis of the raw numbers used to produce the percentages in Table 2 show a statistically significant difference (at the traditional p=0.05 criterion) between the percentages within the "Right" and "Wrong" rows of each column. That is, there is a statistically significant difference in the performance of students on the week 5 test questions, and also on the end of semester exam questions, depending upon whether the students answered Week 3 Question 4 correctly or incorrectly.

It is remarkable that performance on a simple explanation question in week 3 results in a consistent, statistically significant difference in performance in other tasks for the remainder of the semester – problems with relational reasoning start early and persist.

The difference in performance on the week 5 "write a swap" task (i.e. Column A, 57% for wrong vs. 92% for right) is consistent with much of the literature in cognitive psychology. A student who can explain swapping at week 3 remembers that code as a meaningful "chunk". A student who cannot explain that code struggles to remember three separate lines of code.

4.2 Results for Week 5 Write a Swap

Table 3 shows the percentage of students who correctly answered questions from the week 5 test and the end of semester exam. These percentages are broken into two rows, according to whether students correctly answered Question 1 of the Week 5 test, "write a swap". As with Table 2, a chi square analysis showed a statistically significant difference between the "Right" and "Wrong" percentages of each column.

Again, it is remarkable that performance on a simple writing task in week 5 results in a consistent, statistically significant difference in performance on each of the exam questions, especially the dramatically differing performance on writing code to reverse a string (Column D, 27% for wrong vs. 70% for right).

Column A in Table 3 adds to the evidence from BRACElet studies that code writing and code explanation are closely linked cognitive skills. The students who could write the swap code at week 5 (i.e. the row beginning "Right") performed much better on the other week 5 task, where they had to explain some code (Column A, 14% for Wrong vs. 61% for Right). This result again demonstrates that the ability to "chunk" code into meaningful pieces is important in both writing code and explaining code.

Week 5 explain a sort of three variables	Week 5	End of Semester Exam		
	Column A	Column B	Column C	Column D
	write a swap	MCQ, explain product of even nums	trace a swap	write the reverse of a string
Wrong (n = 43)	56%	65%	84%	44%
Right (n = 40)	93%	88%	95%	75%
$\chi 2$ test	p = 0.001	p = 0.02	p = 0.1	p = 0.01

Table 4: The performance of students, broken down according to the week 5 explanation question (n=83).

Prior Programming Experience?	Week 3	Week 5		End of Semester Exam		
	Column A	Column B	Column C	Column D	Column E	Column F
	explain a swap	write a swap	explain a sort of 3 variables	explain product of even nums	trace a swap	write reverse of a string
No (n = 27)	52%	70%	37%	70%	78%	56%
Some (n = 21)	43%	67%	57%	81%	90%	57%
Yes (n = 11)	27%	73%	45%	64%	100%	55%
From Table 1 (n = 83)	47%	73%	48%	76%	89%	59%

Table 5: The percentage of students who answered each question correctly, based on their prior background in programming (n=59, as 24 of the 83 students did not respond to the survey)

4.3 Results for Week 5 Explain a Sort

Table 4 shows the percentage of students who correctly answered questions from the week 5 test and the end of semester exam. These percentages are broken into two rows, according to whether students correctly answered the Week 5, Question 3 explanation question. As with Tables 2 and 3, a chi square analysis showed a statistically significant difference, at the traditional p=0.05 level, between the percentages within each column, except for "trace a swap" (column C).

4.4 Results for End of Semester Trace a Swap

The results in the "Right" and "Wrong" rows of both Table 2 column D and Table 3 column C (both columns for "trace a swap") show a statistically significant difference. For example, Table 3 column C shows that 97% of students who could write a swap in week 5 could successfully trace the swap code at the end of semester, compared to only 68% of students who could not write the swap in week 5.

Given that all n=83 students in this study had passed a screening test where they successfully answered four tracing problems, why should a tracing problem in the final exam present a problem? Our explanation to that question is as follows. Tracing is an error prone activity. The students who were able to explain the swapping code in week 3, or who could write the swapping code in week 5, were more likely to recognize similar code in the final exam. Consequently, those students might have been able to determine the answer to this question in the final exam without having to trace the code, or at least they could have verified their trace by comparing the result to what they thought it should be. However, the other students (i.e. those who were not able explain the swapping code in week 3, or who could not write the swapping code in week 5) would have been less likely to recognize that the code was swapping the values of two variables. Such students had no alternative but to derive the answer by tracing the code, and they had no means of checking their answer, other than by tracing the code again.

4.5 Prior Knowledge

To assess whether prior programming experience may have been a factor in the above results, we analysed the responses to a survey that the students completed at the beginning of the semester. The survey contained the following questions:

- Have you ever written a computer program before? (Yes, No)
- If you answered "Yes" to the above question, in which language or languages have you written computer programs? (Free form answer)
- With respect to programming, attempt to explain what a variable is. (Free form answer)
- With respect to programming, attempt to explain what a function or a method or a procedure is. (Free form answer)
- With respect to programming, attempt to explain what a parameter or argument is. (Free form answer)

On the basis of the answers to the above survey questions, one of the authors classified all the students into 1 of 3 categories:

- "No" – the student indicated they had not programmed before and did not know what variables, methods and parameters were.

- "Some" − either the student indicated they had not programmed before but gave good answers regarding variables, methods and parameters OR the student indicated they had programmed before but could not answer all other questions; usually the parameter question was the problem.
- "Yes" − the student indicated they had programmed and gave good answers for the other questions.

Table 5 describes the percentage of students who answered the test and exam questions correctly, broken down according to the above three categories of prior programming experience. Chi square analysis of the raw numbers used to produce each column of Table 5 showed no statistically significant differences (at the traditional p=0.05 criterion) between the percentages shown within each of those columns. We therefore conclude that prior programming experience is not a confounding factor in the results we have reported.

5 Discussion: To Read, Write and Understand

5.1 Statistics and Causation

We wish to stress that we are not claiming that the ability to write code is dependent upon the ability to explain code. To do so would be to make a well-known fallacy of statistical reasoning commonly stated as "*correlation does not imply causation*". To use a frivolous example sometimes used in introductory statistics lectures, there may be a statistical relationship between ice cream sales and deaths from drowning, but that is because both are linked by hot weather. More formally, two statistical variables may be related because both variables depend upon a third variable.

A possible third variable that links code writing and code explaining is the ability to understand and/or reason about code. Research on the psychology of programming has demonstrated that, as expertise develops, a programmer's knowledge is organized into more abstract, flexible forms, which would benefit both code writing and code explaining (Adelson, 1984; Corritore & Wiedenbeck, 1991; Fix, Wiedenbeck & Scholtz, 1993; Mayer, 1981; Shneiderman & Mayer, 1979; Soloway, 1986).

5.2 Pedagogical Implications

If understanding and/or reasoning about code is the third variable upon which both code writing and code explaining depend, then the crucial pedagogical question is as follows:

> *How can we most efficiently develop our students' capacity to understand and/or reason about code*?

5.2.1 Learning by Code Writing

Is writing code the most efficient way to learn how to understand and/or reason about code? Clearly, students must write some code, but current pedagogical practise emphasises code writing to such an extent that almost all the active learning exercises we give our students (i.e. laboratory exercises and assignments) require our students to write code. Is fighting the compiler the most time efficient way of improving student understanding of code? Perhaps the most efficient way is a judicious mix of having students write code and having them read code (and testing their ability to read via tasks such as explain-in-plain-English).

5.2.2 Roles of Variables

If lecturers are to teach relational reasoning explicitly, and if lecturers are going to set and grade students on exercises where the students must read and understand code, then we need a vocabulary for relational reasoning.

One promising vocabulary is "*roles of variables*" (Ben-Ari & Sajaniemi, 2004; Kuittinen & Sajaniemi, 2004; Sajaniemi, 2010). These are a dozen categories for the purpose of a variable in a piece of code. Three of these roles are:

- ***Stepper*:** is defined as being "*a data entity stepping through a succession of values that can be predicted as soon as the succession starts*". This role is illustrated by the for-loop control variable "`i`" in the explain-in-plain English question on the first page of this paper.
- ***One-way flag*:** is defined as being "*a two-valued data entity that cannot get its initial value once its value has been changed*". This role is illustrated by the variable "`bValid`" in the explain-in-plain English question on the first page of this paper.
- ***Temporary***: is defined as being "*a data entity holding some value for a very short time only*". This role is illustrated by the variable "`temp`" in the code on the second page of this paper, which is code for swapping the values of two variables.

Lecturers could teach these roles, and explain code in terms of these roles. Students could be graded on exercises where they identify the roles of variables in a piece of code, perhaps as part of an explain-in-plain-English question. Our intuition is that a student who can identify the roles of all the variables in a piece of code is close to explaining what the code does (but that is a conjecture that would make for interesting future work).

6 Conclusion

Understanding three assignment statements, that swap the values in two variables, is not rocket science. Neither is writing that same code. However, we have shown that, in week 3 of semester, half of the students in our sample have a problem with understanding such a simple piece of code, and two weeks later one half of those students cannot write that same code. Furthermore, as a group, these students who could not answer those questions in weeks 3 and 5 performed relatively worse on programming tasks in the final exam. Thus, from the very early stages of the semester, the students begin to separate into two groups. The students in one group tend to think relationally about code, of their accord. The students in the other group do not tend to think relationally about code. Early detection and treatment of those students in the second group may improve failure rates.

We are not advocating that thinking relationally is an innate skill. Instead, we believe that current pedagogical practice does not help novice programmers learn to think relationally. Today, learning to think relationally about code is an implicit part of the curriculum of

programming. Some of our current students succeed in teaching themselves that implicit part of the curriculum, but many do not. We need to develop pedagogical techniques that transform this implicit component of the curriculum into an explicit part of the curriculum.

Finally, we urge the reader to either use our two in-class tests, or design their own tests that are more to their liking, and collect data from their own class. Not only might the results illuminate the reader's thinking about their own teaching, but replications of our study will determine whether the statistical relationships we have found are widespread, or are the result of some relatively unusual aspect of our teaching environment.

Acknowledgements

Raymond Lister's participation in this work was partially funded by an Associate Fellowship awarded by the Australian Learning and Teaching Council.

References

Adelson, B. (1984) When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **10**(3), 483-495.

Ben-Ari, M. and Jorma Sajaniemi, J. (2004) *Roles of variables as seen by CS educators*. 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE). pp. 52-56. http://doi.acm.org/10.1145/1007996.1008013

Biggs, J. B. and Collis, K. F. (1982): *Evaluating the quality of learning: The SOLO taxonomy* (*Structure of the Observed Learning Outcome*). New York: Academic Press.

Corney, M., Teague, D., Thomas, R. (2010) *Engaging Students in Programming*. Twelfth Australasian Computing Education Conference (ACE 2010), Brisbane, Australia, January 2010. CRPIT, **103**. Tony Clear and John Hamer, Eds., ACS. pp. 63-72. http://crpit.com/confpapers/CRPITV103Corney.pdf

Corritore, C. & Wiedenbeck, S. (1991) What Do Novices Learn During Program Comprehension? *Int. J. of Human-Computer Interaction*, **3**(2), 199-222.

Fix, V., Wiedenbeck, S., and Scholtz, J. (1993) *Mental representations of programs by novices and experts*. In INTERACT '93 and CHI '93 Conferences on Human Factors in Computing Systems. pp. 74-79. http://doi.acm.org/10.1145/169059.169088

Kuittinen, M, and Sajaniemi, J. (2004) *Teaching Roles of Variables in Elementary Programming Courses*. 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE). pp. 57–61.

Lister, R. And Edwards, J. (2010) *Teaching Novice Computer Programmers: bringing the scholarly approach to Australia − A report on the BRACElet project*. Australian Learning and Teaching Council, Sydney, Australia. ISBN: 1-921856-02-5. Downloadable from http://www.altc.edu.au/altc-associate-fellow-raymond-lister

Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006) *Not seeing the forest for the trees: novice programmers and the SOLO taxonomy*. 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE), 118-122. http://doi.acm.org/10.1145/1140123.1140157

Lister, R., Fidge C. and Teague, D. (2009) *Further evidence of a relationship between explaining, tracing and writing skills in introductory programming*. 14th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE), pp 161-165. http://doi.acm.org/10.1145/1595496.1562930

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., and Thompson, E. (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bull.* **41**, 4 (Jan.), pp. 156-173. DOI= http://doi.acm.org/10.1145/1709424.1709460

Lopez, M., Whalley, J., Robbins, P., and Lister, R. 2008. *Relationships between reading, tracing and writing skills in introductory programming*. Fourth International Workshop on Computing Education Research (ICER), Sydney, Australia. pp. 101-112. http://doi.acm.org/10.1145/1404520.1404531

Mayer, R. (1981) The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, **13**, 1 (March), pp. 121-141. http://doi.acm.org/10.1145/356835.356841

Sajaniemi, J. (2010) Roles of variables. http://cs.joensuu.fi/~saja/var_roles/ [Accessed Nov 2010.]

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., Whalley, J. (2008) *Going SOLO to Assess Novice Programmers*. 13th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE), Madrid, Spain. pp. 209-213. http://doi.acm.org/10.1145/1384271.1384328.

Shneiderman, B. & Mayer, R., (1979) Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information*, **8**(3), pp. 219-238.

Soloway, E. (1986) Learning to program = learning to construct mechanisms and explanations. *CACM,* **29**, 9 (Sep), 850-858. http://doi.acm.org/10.1145/6592.6594

Venables, A., Tan, G., and Lister, R. (2009). *A closer look at tracing, explaining and code writing skills in the novice programmer*. Fifth International Workshop on Computing Education Research (ICER), Berkeley, USA. pp. 117-128. http://doi.acm.org/10.1145/1584322.1584336

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P.K.A. and Prasard, C. (2006). *An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies*. Eighth Australasian Computing Education Conference (ACE2006), Hobart, Australia. CRPIT, **52**, pp. 243-252. ACM International Conference Proceeding Series, Vol. 165. http://crpit.com/confpapers/CRPITV52Whalley.pdf

INB104 Test 1, [Sem 1, 2010 Week 3], page 1

Student's Name _____ Student's Number _____

For all questions in this test, you may write down any working out on this test paper, except in the answer boxes. Write ONLY your answer in the answer boxes.

Q1. In the boxes provided below, write the values in the variables after the following code has been executed:

```
r = 2
s = 4
r = s
```

The value in `r` is [ ] and the value in `s` is [ ]

Q2. In the boxes provided below, write the values in the variables after the following code has been executed:

```
p = 1
q = 8

p = q
q = p
```

The value in `p` is [ ] and the value in `q` is [ ]

Q3. In the boxes provided below, write the values in the variables after the following code has been executed:

```
x = 5
y = 3
z = 7

x = z
y = x
z = y
```

The value in `x` is [ ] the value in `y` is [ ]

and the value in `z` is [ ]

The rest of the test is on the other side of this piece of paper …

INB104 Test 1, [Sem 1, 2010 Week 3], page 2

Student's Name _____ Student's Number _____

For all questions in this test, you may write down any working out on this test paper, except in the answer boxes. Write ONLY your answer in the answer boxes.

This is page 2 of the test. The rest of the test is on the other side of this piece of paper.

Q4. The purpose of the following three lines of code is to swap the values in variables a and b:

```
c = a
a = b
b = c
```

The three lines of code below are the same as the lines above, but in a different order:

```
a = b
b = c
c = a
```

In one sentence that you should write in the box below, describe the purpose of those second set of three lines. **NOTE:** Tell us what the second set of three lines of code do all by themselves. Do NOT think of those second three lines as being executed after the first three lines of code.

[ ]

Q5. In one sentence that you should write in the box below, describe the purpose of the following three lines of code for any set of values stored in variables i, j and k:

```
j = i
i = k
k = j
```

[ ]

***** End of Test *****

INB104 Test 2, [Sem 1, 2010 Week 5], page 1

Student's Name _____ Student's Number _____

For all questions in this test, you may write down any working out on this test paper, except in the answer boxes. Write ONLY your answer in the answer boxes.

Q1. Suppose you have two integer variables, called `p` and `q`. **In the box below write code to swap the values in those two variables**. You may declare and use any extra variables required to make the swap. Give each extra variable a meaningful name that reflects its purpose.

Q2. This question is about the following code, where the variables `p`, `q`, `r` and `s` all have integer values:

```
if (p < q):
    if (q > 4):
        s = 5
    else:
        s = 6
```

Assume that, **before** the above code is executed, the values in the four variables are:

```
p = 1      q = 2     r = 3     s = 4
```

After the codes is executed, the value in variable `s` is

INB104 Test 2, [Sem 1, 2010 Week 5], page 2

Student's Name _____ Student's Number _____

For all questions in this test, you may write down any working out on this test paper, except in the answer boxes. Write ONLY your answer in the answer boxes.

Q3. If you were asked to describe the purpose of the code below, a good answer would be "*It prints the smaller of the two values stored in the variables* `a` *and* `b`".

```
if (a < b):
    print a
else:
    print b
```

In one sentence that you should write in the empty box below, describe the purpose of the following code.

Do **NOT** give a line−by−line description of what the code does. Instead, tell us the purpose of the code, like the purpose given for the code in the above example (i.e. "*It prints the smaller of the two values stored in the variables* `a` *and* `b`").

Assume that the variables y1, y2 and y3 are all variables with integer values.

In each of the three boxes that contain sentences beginning with "`Code to swap the values …`", assume that appropriate code is provided instead of the box – do **NOT** write that code.

```
if (y1 < y2):
```

```
Code to swap the values in y1 and y2 goes here.
```

```
if (y2 < y3):
```

```
Code to swap the values in y2 and y3 goes here.
```

```
if (y1 < y2):
```

```
Code to swap the values in y1 and y2 goes here.
```

```
print y1
print y2
print y3
```

***** End of Test *****