Evaluating a New Exam Question: Parsons Problems

Paul Denny
Computer Science Dept.
University of Auckland
New Zealand
+64 9-373-7599
paul@cs.auckland.ac.nz

Andrew Luxton-Reilly
Computer Science Dept.
University of Auckland
New Zealand
+64 9-373-7599
andrew@cs.auckland.ac.nz

Beth Simon
Computer Science and Engr. Dept.
University of California, San Diego
La Jolla, CA, USA
+01 858-534-5419
bsimon@cs.ucsd.edu

ABSTRACT
Common exam practice centres around two question types: code tracing (reading) and code writing. It is commonly believed that code tracing is easier than code writing, but it seems obvious that different skills are needed for each. These problems also differ in their value on an exam. Pedagogically, code tracing on paper is an authentic task whereas code writing on paper is less so. Yet, few instructors are willing to forgo the code writing question on an exam. Is there another way, easier to grade, that captures the "problem solving through code creation process" we wish to examine? In this paper we propose Parson's puzzle-style problems for this purpose. We explore their potential both qualitatively, through interviews, and quantitatively through a set of CS1 exams. We find notable correlation between Parsons scores and code writing scores. We find low correlation between code writing and tracing and between Parsons and tracing. We also make the case that marks from a Parsons problem make clear what students don't know (specifically, in both syntax and logic) much less ambiguously than marks from a code writing problem. We make recommendations on the design of Parsons problems for the exam setting, discuss their potential uses and urge further investigations of Parsons problems for assessment of CS1 students.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *curriculum, computer science education.*.

General Terms
Design, Human Factors

Keywords
Exam questions, Parsons Problems, CS1, assessment, tracing, code writing.

ICER'08, September 6–7, 2008, Sydney, Australia.

1. INTRODUCTION
Exams are a commonly used tool by which student performance in beginning programming courses is examined. Though often used in conjunction with other assessments such as "out of class" programming assignments and labs – most institutions place significant value on written, paper exams for allowing student progression through the curriculum. As such, studies of exam questions should be of critical interest to faculty. In 2001, the McCracken ITiCSE working group [6] showed poor first-year student skills at writing code (across 216 students at four universities). Though this study was not an exam setting study, it still gave educators concern over student code writing abilities. The Leeds ITiCSE working group [4] in 2004 followed up by looking at two types of "tracing" multiple choice exam questions which focus on code reading and understanding (with 941 students at 12 institutions). These were selected both as common exam question types and as a possible cognitive pre-cursor to code writing ability. The Leeds questions fell into two categories: reading (and tracing) code execution (with loops and, often, arrays) and code line selection (e.g. where students select from a set of possible for loops to complete given code). While the full argument of whether code reading is cognitively a necessary pre-cursor of code writing was not fully addressed by this work, student performance on these multiple choice questions was much better than on the McCracken code writing study.

The BRACElet project has continued the investigation of exam questions for CS1 [1, 5, 12]. Previous work has considered the level of learning evidenced in reading code – by marking student answers to questions of the form "Explain in plain English what this code does." Code tracing questions typically fall into the lower categories in Bloom's taxonomy, while code writing questions are usually categorized as requiring higher order cognitive skills [11]. Explain in plain English questions are an effort to span the assessment range between mechanically tracing computational processes and the marshalling of both creative and Java language knowledge to solve a problem by producing code in a programming language. However, "Explain in plain English" still weighs more heavily on the trace and comprehend side of the trace/code balance. In this work, we explore the other code/comprehend side through scaffolded code creation.

In this study we explore a new type of exam question – the Parsons problem [8]. Previously developed to help students acquire competence with the structural syntax of programming, we are interested in the ability of the problem type to tell us

something we can't get from code tracing or open-ended code writing questions in an exam setting. Parsons problems, in essence, provide all the code required to solve a given problem – but require the students to order (and possibly select then order) the lines of code to form a correct solution. We report here two studies on Parson's problems for exams. First is an open-ended qualitative study of the potential of Parsons problems in exam-type settings (with pencil and paper) by interviewing 13 students who recently completed a CS1. We sought to determine how they approached problems requiring code re-ordering, their experiences with 5 different variants of Parsons problems to identify desirable problem design characteristics, and to get a general understanding of the differences in skills and experience between Parsons problems and code writing and tracing. This study sought to help us develop a format and design for Parsons problems that would have students engaging problem solving skills with code (not simply employing lexical and formatting rules) while minimizing irrelevant cognitive load.

Using this knowledge, we designed a second quantitative study to document student performance on Parsons problems in conjunction with code writing and tracing on a CS1 final exam. All three of these questions were designed to exercise approximately the same programming understanding – requiring the use of for loops to calculate a value based on array scanning. After marking the exams for research purposes, we find that ability in Parsons and code writing is notably correlated (Spearman's r^2 = .53) and that the other two combinations of questions are not well correlated (Parsons-tracing Spearman's r^2 =.19 and code writing-tracing Spearman's r^2 =.37). We also demonstrate the clarity with which one can determine what specific logic and syntax issues students have not mastered. Furthermore, these Parsons questions can be marked quickly with less variation between markers than similar code writing questions. We urge the community to continue to investigate code tracing, "explain in plain English", Parsons and code writing problems as assessment variations which give us more robust understandings of students' abilities in CS1.

2. BACKGROUND

This is based most directly in the work of the BRACElet project which has looked at a number of issues surrounding CS1 exam questions and programming knowledge. BRACElet consists of a group of CSEd researchers, based primarily in Australasia, engaged in ongoing research into student understanding through analysis of examination questions and student responses to these questions. BRACElet grew out of two seminal studies by ITiCSE working groups, as described in the introduction.

The term "Parsons' programming puzzle" was introduced in [8] in the context of a software tool which allows students to interactively piece together programming solutions from fragments. A similar software tool named CORT has been used to support the completion method of instruction [2]. A partially completed solution is provided and students are expected to select lines of code that correctly complete the solution and arrange them appropriately. The concept of writing code by selecting from provided fragments can also be applied effectively in traditional written exams. We have chosen to designate exam questions of this style as "Parsons problems". There is, however, great flexibility in how these questions can be designed, including the types of fragments from which to select, and how much structure of the solution is provided in the question. At a recent BRACElet meeting in 2007, we analysed the complete set of student responses to a Parsons problem used in a final exam at the Auckland University of Technology. The complete block structure of the solution was included in the question, and the provided lines of code simply needed to be reordered and inserted into that structure. Students were left with few opportunities for making errors and as a result the marks for this question were extremely high, leaving the responses unsuitable for analysis. In this paper, we considered a number of possible styles for Parsons problems, and are able to make a recommendation of the style that we feel is most appropriate for use in written exams.

The authors' attempts to find similar "rearrange correct components to produce a complete answer" from other disciplines were largely unsuccessful. An area of possible potential similarity is with mathematical proofs. Proofs are made up of a number of smaller steps, each of which could be subjected to a small "syntax-style" mistake or a larger "logic-style" mistake. Students have been shown to have difficulty with notational (e.g., syntax-related) issues in proofs [9]. Additionally, student ability to "find bugs" in proofs has also been explored [10]. One could imagine that a very beginning proof-learning strategy would be just to have students re-order correct lines to form a correct proof. In physics, the most closely related work is in "ranking task exercises", of which there are two workbook-style textbooks based on physics education research [3, 7]. In this case, students are provided "a question with several contextually similar situations [which] differ in the value of one or more physical quantities" [7]. For example, one problem asks students to rank how high eight arrows would fly, with each arrow having a different mass and initial velocities. In this case, students are asked to "rank" not to form a single solution (as with a code snippet) but to compare between scenarios. This is most closely related to our Parsons problems when we ask students to choose between two lines of code that differ logically.

3. METHODOLOGY

In this section we report on the two components of our research methodology. The first study uses a set of student interviews to assess student experience with a set of potential exam-type Parsons problems. The second evaluates a specific Parsons problem on an exam in comparison with standard code tracing and code writing questions.

3.1 Study 1: Interviews on Parsons Variants

The work reported here began with a desire to understand how students were likely approaching Parsons problems in an exam-style pencil and paper type setting. To this end, a series of interviews with 13 North American undergraduates were conducted. Students were approximately four weeks into a second computing course following a Java, objects-first CS1. Students were recruited through an in-class announcement and email which specifically stated that we wanted to talk to students who did well and not so well in CS1. Anecdotal evidence indicates that we were successful in attracting at least a few non-top performing students. Six of those interviewed were women. The first six interviews followed the same format which a) compared student Parsons ability to code writing ability and b) investigated five variants of Parsons problems. Interviews 7-13

differed in that they a) compared student Parsons ability to code tracing ability and b) refined two of the Parsons variants and eliminated a third.

3.1.1 Protocol Format

The protocol format differed slightly in the two sets of interviews. In the first set, students were informed that we were considering a new exam question format that differed from the standard code tracing and code writing questions. They were informed that we would show them five variants (described in the next section) with a really easy problem (print out the first five even numbers starting with two) and let them "get used" to the problem types. We asked them to think aloud while working and in between each problem variant, we asked them "what was different about this one" and occasionally asked "how was this harder or easier" when necessary.

After the warm up, we showed the students four looping problem descriptions which included: find the average of elements in a sub-range of an array, print a chequered pattern of stars, create a new word with only the consonants of the inputted word, and find the maximum two values from a set of 10 inputted values. We asked the students to indicate which of these they thought they could write correct code for (on paper with pencil) in 3-5 minutes. We also asked them to rank on a scale of 1-5 how hard they thought the problem was. We then explained to the students that we specifically wanted to identify a coding problem that was slightly too difficult for them to solve quickly. We felt it was important to clarify this expectation to the students in order to put them at ease as we didn't want them to feel frustrated or stressed at not being able to solve a problem during an interview.

We explained that we were planning to give them a series of the Parsons variants from harder to easier and if they "got it" on the first variant, then we wouldn't really find out much. In addition, we stated that while we would be limiting the time allowed for this exercise, we felt confident in the student's ability to solve the problem without such time constraints.

The interviewer then selected a problem based on the student difficulty ranking and their interviewer's limited assessment of the "easy problem" think aloud and asked the student to try to write the code for that problem. In most cases the students stumbled after a few minutes – in the cases where they completed an answer quite easily, we congratulated the student but expressed our regret at not selecting a hard enough problem and asked them to code again with one of the harder problems.

In the next portion, we provided the student with 5 Parsons variants of the problem they were unable to code – with decreasing difficulty. Students were allowed to work 3-5 minutes on a variant – or until the interviewer saw them struggling – often with an issue that they might be "helped with" in the next easier variant. Think alouds continued in this section with the same reflection questions asked between each variant. Some students would complete the problem on the second to easiest variant and were just asked to talk about how the easiest version differed.

For the final portion of the interview, we wanted to explore how students coded, after working (and usually solving) a Parsons problem. After a distraction task (having the student talk about their major, their interest in CS and/or their usual study habits in CS), we then asked students to try again to write the code for the problem on which they had been working.

The second set of interviews followed a very similar protocol, except that instead of being asked to write code (at the beginning and end) they were asked to trace code. Obviously, we could not give students the exact code to trace which solved the problem they had just selected to work on (they could "know" the answer without tracing code execution). So we developed and gave them a "mystery code" with meaningless variable names that was a direct corollary (in terms of specific programming knowledge needed) to the problem they had chosen to solve. That is if the problem required arrays and if statements, we developed a corollary that used arrays and if statements in very similar ways. If a problem required looping over characters in a String, we gave a tracing corollary that also required that.

3.1.2 Parsons Variants

We considered five initial variants of Parsons problems. In one set (two variants) we provide only the exact lines of code needed to solve the problem (though they are jumbled in order) in the other set (three variants) a superset of the required lines is provided – with most lines in the code having two options presented (often the method header would have only one option). In the exact line set the first variant (easiest) involved having a scaffolded set of braces and boxes to indicate indentation, structure, and number of lines in a structure. The harder variant simply provided blank space and students were instructed to fill in their own braces.

The second set had two variants (scaffolded and not) similar to the exact line set, but with the superset of code line options presented with paired options listed in "order" (that is, if there was an option for that line, it was next to the correct line in the jumbled order). However, the grouping of these lines was not specifically designated to students – and the numbering of lines (with letters) was simply chronological (A-Z). In the third variant for this set the lines were completely jumbled so that "line options" were not next to each other.

In the second set of interviews (7-13), we updated our Parsons variants based on student comments and difficulties. We discarded the completely jumbled version as being completely unreasonable. We also renumbered the superset versions so that a) there was exactly 2 options for each line of code and b) pairs were made explicit through line numbering such as A1, A2, B1, B2, etc. We stated explicitly that a correct solution would involve picking one option from each pair of lines. We did not, however, state that only one set of options led to a correct solution.

3.2 Study 2: Parsons in a CS1 Examination

In this section, we present three questions that were included in an exam taken by 74 students and discuss the development of an appropriate rubric that was used to mark and compare the student responses to these questions. The exam was the final assessed activity of the CS1 course at the University of Auckland in the summer semester of 2008, and carried a 60% weighting in calculating the final grade for each student. The exam was comprised of 11 questions, including a code writing question, a code tracing question and a Parsons problem. Each of these three questions required iterating over an array (in one case, indirectly, as a String).

The style of the Parsons problem was chosen based on the feedback we received from the interviews described in the

```
    return result;
    return word;

    String result = "";
    String result;

    if (word.charAt(i) == 'a')
    if (word.charAt(i) != 'a')

    for (int i = 0; i < word.length(); i++)
    for (int i = 0; i < word.length; i++)

    result = result + word.charAt(i);
    result = word.charAt(i);

    private String removeAllAs(String word)
    private String removeAllAs(word)
```

Figure 1. The Parsons Problem Exam Question

previous section, and consisted of a paired superset of the required lines of code. These lines were given in the order and using the pairings shown in Figure 1.

Students were required to select the correct line of code from each pair, and place the selected lines of code in the correct order to define a Java method for removing all occurrences of the character 'a' from a String.

This style of question makes distinguishing syntax errors from logic errors clear. Selecting the incorrect line from the second, fourth, or sixth pairs would prevent the code from compiling and hence were classified as syntax errors. Selecting the incorrect line from the first, third, or fifth pairs would generate an incorrect result, and were therefore classified as logic errors. We will use the terms "Return", "Initialisation", "Conditional", "Loop", "Accumulator" and "Header" to refer to the line options.

The code tracing question required the students to determine the effect on a given array, passed as a parameter to a method called `doSomething()`. Specifically, they were asked for the output from the following code (the entry point is the `start()` method):

A correct trace of this code would show that the method reverses the order of the array elements. In addition to carefully tracing through the loop, students needed to recognise that an array is passed by reference to a formal parameter.

For the code writing question, the students were asked to write a method which is passed an array of Strings, and which returns a

```
public void start() {
        int[] numbers =
                    {17, 2, 9, 8, 11, 6, 1, 10, 5};
        doSomething(numbers);

        for (int i = 0; i < numbers.length; i++) {
                System.out.print(numbers[i] + " ");
        }
}

private void doSomething(int[] nums) {
        int temp;
        for (int i = 0; i < nums.length/2; i++) {
                temp = nums[i];
                nums[i] = nums[nums.length-1-i];
                nums[nums.length-1-i] = temp;
        }
}
```

Figure 2. The Code Tracing Exam Question

```
private String getFirstLetterString
                                (String[] words ) {
        String firstLetters = "";
        for (int i = 0; i < words.length; i++) {
                firstLetters = firstLetters +
                                words[i].charAt(0);
        }
        return firstLetters;
}
```

Figure 3. The Code Writing Exam Question

new String consisting of a concatenation of the first character from each of the Strings in the array. One possible correct answer to this question is shown in Figure 3.

3.2.1 Marking rubrics

The exam was initially marked by the teaching staff of the course. While this provided a good indication of the spread of marks, the marking schemes that were used were somewhat subjective, particularly for the code writing question. A negative marking scheme, where marks were deducted for errors, was employed for code writing answers that were nearly correct. However, for answers that were very incomplete or badly incorrect, a positive marking scheme was used in which marks were awarded for any parts of the answer that were correct. For other answers, some combination of positive and negative marking was used. While this is quite subjective, it is also fairly efficient and, anecdotally, it seems to be a fairly common way of marking exams where there is pressure to calculate grades in a timely fashion.

For the purposes of this study, we set about defining our own marking scheme for each question. We chose to employ a negative marking scheme, by defining the types of errors for which marks would be deducted.

Parsons problem

The Parsons problem consisted of 6 pairs of statements, and we deducted 1 mark for each incorrectly chosen line from the pairs. We also looked at the ordering of the lines, and deducted 2 marks if more than 2 lines were out of place and we deducted 1 mark if up to 2 things were out of place. Finally, we deducted 1 mark if the student incorrectly used opening and closing braces, or did not include them where necessary.

Each author marked every answer to the Parsons problem according to this rubric, which was a fairly straightforward process. Our intial coding achieved an 92% inter-rater reliability rating which only differed in ordering points. The few minor discrepancies were attributed to an error in applying the rubric and were corrected.

Writing

For the writing question, we allocated 3 marks for syntax and 8 marks for logic. A correct answer consists of a loop, an

Table 1. Marking rubric for the Parsons problem (both syntax and logic components) out of 9 marks

Syntax (3)	*Logic (6)*
Header (-1)	Conditional (-1)
Initialisation (-1)	Accumulator (-1)
Loop (-1)	Return (-1)
	Braces (-1)
	Ordering (-2)

accumulator, and a return statement and our logic rubric uses these components.

The syntax rubric was developed empirically from the common mistakes of students. However, an issue arose with this scheme. If very little code had been written, it was likely that few marks could be deducted. This is a case where, traditionally, a positive marking scheme might have been applied. To penalise answers that were incomplete, we capped the syntax mark based on the number of lines of code that had been written. A summary of the logic and syntax rubric for the writing question is in Table 2.

Defining the rubric for the writing question was more difficult than for the Parsons problem as the open nature of the question meant that there were more possibilities for errors that we needed to consider. We started by marking the first 20 scripts, and then discussed shortcomings of the rubric. After several iterations, we agreed on the rubric described above. Once again, we each marked all of the answers to the code writing question. However, there was still significant disagreement over the marking. We achieved an inter-rater reliability of 68% on marking even after our extensive discussion. On 90% of these marks we differed by at most one point. We resolved these discrepancies, through discussion focussed on how the rubric should be interpreted in borderline cases rather than simple errors in applying the rubric that were more common when marking the Parsons problem.

Table 2. Marking rubric for code writing out of 11 marks.

Syntax (3)	Missing/wrong types in local declarations (-1)		
	Missing/Wrong return type (-1)		
	Missing/Wrong parameter type, usually [ ] notation (-1)		
	Missing index into array variable (-1)		
	Bad array length syntax, using words.length() (-1)		
	Cap on marks by number of lines of code		
	1 line of code (only the header)	0	
	2 lines of code	1	
	3 lines of code	2	
	4 lines of code or more	3	
Logic(8)	**Loop (2)**	Off by one error, e.g. using <= or starting index at 1 (-1)	
		Incorrect loop condition (-1)	
		Significant error composing the loop (-2)	
	Return (2)	Incorrect placement of return statement, e.g. inside the loop (-1)	
		Missing return statement (-2)	
	Accumulator (4)	Missing/incorrect initialisation of accumulator, e.g. initialisation inside the loop (-1)	
		Minor error in accumulation statement, e.g. concatenation in the wrong order, or wrong index used for the array access (-1)	
		Significant error in accumulation statement, e.g. assignment instead of concatenation, or accumulation outside loop (-3)	
		Missing or nonsensical accumulator (-4)	

Table 3. Marking rubric for code tracing out of 5 marks.

Answer classification	Marks (5)
Perfect	-0
Right thing to half the elements Only one half of the sequence had been correctly reversed (first half, usually)	-1
Off by one Initial or terminating iteration of the loop was missed	-1
Something to half the elements Modified half of the array, but not by reversing the elements	-4
Lost Incorrect and could not be classified	-5
Same as original array	-5

Each student answer was therefore a listing of a sequence of numbers, and as such it was sometimes difficult to determine the intermediate steps that students had taken in arriving at their solution. For marking this question, we defined six categories into which answers could be classified as shown in Table 3.

As an example of "Something to half the elements," a student had assigned the same value to each element in the first half of the array. We felt this was deserving of some credit, as it showed the student at least understood that the loop iterated over only half of the array.

It was possible for an answer to be classified into more than one category. For example, if a student reversed the first half of an array, but missed the first element, that answer would be classified into both "Right thing to half the elements" and "Off by one". In fact, an "Off by one" classification was always combined with either "Perfect" or "Right thing to half the elements".

As before, each author individually marked each answer to this question, and achieved a 93% inter-rater reliability in the classifications.

The difficulty in defining the rubric for the tracing question was that the mark assigned to each student was based solely on the single line of output they provided. Sometimes there was evidence that the student had systematically traced through the code, based on annotations and diagrams drawn by students on the question paper, yet the final output was incorrect. Basing the mark on the final output alone was a shortcoming in the question, and it would have been preferable to require the student to show the output at intermediate steps – perhaps by including a print statement inside the loop.

4. STUDY 1: WHAT DO THESE ASSESS? QUALITATIVE RESULTS

First, we report on the set of 13 interviews we used to identify and illuminate how students experienced and thought about Parsons problems. These interviews were intended to help us design a good Parsons problem for analysis on an actual exam and provide explanations which would allow us to feel more confident about how Parsons problem ability compared to student code writing ability and code tracing ability. Although the evidence here comes from a small number of observations, each student performed between 8-10 Parsons problems and was frequently asked to reflect on various aspects of the process. Students, in general, seemed to take this very seriously and we often found their comments to reflect the interviewer's observations.

4.1 Structural Design Issues which Obscure

Three main design issues became evident from comparison of variant problem types during interviews. Student feedback on these issues was used to inform the design of the exam question, given in Figure 1, on which our quantitative study is based.

Completely jumbled (non-paired) superset of code lines are overwhelming. This observation led to the dropping of our "hardest" variant from the second half of the student interviews. Student comments like "arrrgh!" and clear instances of being overwhelmed were observed when students were presented with a completely jumbled superset of lines to pick from to solve a problem (compounded by single letter variable names). This clearly imposed a cognitive load that seemed to be little related to programming or problem solving knowledge. P12 (who showed excellent programming ability) stated, "writ[ing] the letter -- it's very confusing and tedious."

Students often thought that perhaps there was more than one selection of code lines that might be correct. P10 stated, "it looks like you can get 2 varieties of code that will work and I am trying to decide if that is the case." She then seemed to try to work two possible solutions in parallel for a time. P03 also spent time on this issue. Suffering from the same concern, P02 said "if you were to pick the wrong one, you would eventually get stuck."

In general, with the completely jumbled superset code, students seemed to need to try to guess what the instructor writing the problem might have possibly had in mind – rather than thinking of a solution somewhat related to how they would have solved the problem. Based on the overwhelming difficulties we saw we recommend the design used in our second set of interviews – every line should have two explicit options (perhaps numbered A1,A2 and B1,B2) and students should explicitly be told that there is only one possible selection of lines that is correct. This appeared to allow students to be more focussed on problem solving and less on guessing intention. Note also, that even a 13 line program when doubled to 26 lines completely jumbled can be pretty daunting at a first glance.

Referring to code lines by "letters" adds cognitive load. Our next most ubiquitous comment from students was that "lettering" lines of code so that students wouldn't have to copy lines of code was a huge hindrance. P09 said it well -- "1/2 way through and you can't remember what the letters for the top half stand for." Moreover, P13, P03, P04, and P06 concurred. P01 thought that the instructions to "don't write your own code" meant she "wasn't allowed to" copy down the code lines given (perhaps marks would be deducted for it). Several students caught themselves accidentally copying down the wrong letter – and they spent quite a bit of time shifting their gaze from where they were "writing" their solution and the lines of code they were provided. Even worse, P03 pointed out that the letters would disincline her from checking her answer – not a trait we want. P10 recommended -- "give us lines, not boxes so I can write out the code if I want" on the structured problems. Others copied out the lines on scratch paper because it was "hard to see the code because you are not looking directly at it" [P11] and also because "that's a lot of visual processing required" [P10]. They liked writing out the code so they could see what it looked like – this is discussed further in Section 4.3.

Indications of structure can be used to make an "easier" problem on an exam. Almost all students agreed that structure (in the form of provided braces ({}), indentation, and indications of the number of lines of code that belonged in the solution made a problem easier. This was strongly supported by interviewer observation. P09 said "with the braces it's easier to find your mistakes" and P11 said it "helped me realize" that he needed two statements inside an if block. P04 had the exact same problem (as P11) with the same code saying "Boxes helped -- I didn't think to transfer x to a -- because there is an extra line -- so I realized that I needed it." The interviewer saw several similar cases where, because students would get "stuck" working the unstructured variant and then be given the structured version next, students actually discovered their obstacle or mistake by seeing structure and an indication of the number of lines of code that should go in each "section". P06 summarizes by saying "it gives a lot of hint" and contrasting to having to figure out for herself what goes in the loop and what doesn't on the unstructured problems.

Students also commented on how structure allowed them to familiarize themselves with the possible solution they were looking for. P07 used the bracket structure to recall similar problems and consider a pattern he might apply – "recall what kind of structure I have seen before -- and choose from the options." This was helpful for several students -- except when one makes mistakes regarding the information you "gain" from the structure. In P07's case, he assumed (as others did) that a set of {} means there's a loop. Another student, P10, works similarly -- "I'm going to cheat a bit, I see the braces and I think there are two loops. And there are 2 loops here." However, he then struggles because the code actually has a loop with an if statement in it (the two loops were the actual loop header and its pair distracter). It should be noted that, in the CS1 the students took, they saw many examples of ifs with a single associated statement and therefore blocks that did not use curly braces.

Although one student particularly noted the brackets as "intimidating" [P01], this seemed rare. P04 brought together the issue of having a superset of code lines and having structure. He claims that "the structure is useful when there is wrong code. When all the lines have to be used then it's doesn't make a difference" (having structure or not) [P04].

4.2 Comparison to Other Exam Questions

Many students were of the strong opinion that they preferred to write their own code (rather than solve Parsons problems). Often this was expressed as "prefer[ing] to solve things my own way." [P03] P01 expounds, "there are multiple ways to approach it. And multiple ways to get there. That's what throws me off. Writing it on your own you have more leeway." P04 brings in a comparison to other exam question styles saying, "even in multiple choice [exam questions] I would come up with an answer before choosing an option" (something he felt he couldn't do with Parsons).

A few students did recognize the benefit in an exam situation. P11 says, "this tests out problem solving more than syntax." And P12 provides qualified preference saying, "I prefer code writing -- the advantage this has is it gives you the syntax you need." P06 gives the sole expression of Parsons as "pleasurable" for an exam saying, "I like this since I am copying answers" (particularly in reference to a non-superset variant).

When comparing to code tracing exam questions, we found a surprisingly strong uniformity of dislike towards tracing questions on exams. Parsons was considered to be "not so bad" as tracing questions. P11 summarized nicely much of the discussion we heard from students doing code tracing. "Code reading problems on exam -- you have to track all the numbers on paper -- and that's annoying -- loops get tedious. Rearranging code is less tedious -- you are bound to get confused if you do the same thing over 5 times (in tracing)." [P11] P12 says, "in tracing you might make a small mistake it might cost you the mark (on the exam) -- here you are really testing the actual code itself. In the real world, you wouldn't trace the numbers, you'd run the program."

4.3 CS1 Student Mental Models as Evidenced by Discussing Parsons Problems

As part of the think aloud process, students also made meta-comments that were either replicated in many Parsons variants or reflected their comparative experiences with code writing and code tracing. We saw common evidence in three areas that contribute to the knowledge base on novice students' mental models of computation.

Students rely on "seeing code structure". This often came up in conjunction with the cognitive load of the lettered results, but we hear it from almost all students -- there is comfort in "seeing the shape" [P06] of the code. P06 later re-asserts this claim in the context of code writing saying, "if you know the shape of it, then you can do it."

This issue was paramount when students were clearly struggling. Student commentary indicated a pattern of simply trying to apply rules that they have derived from somewhere (where was not clear). P08 indicated that the ordering of code lines to solve a problem was "a matter of speech. I know that you read in variables in the initialization section" (even though the code calls for reading in 10 values in a loop). P10 also leverages "rules" about code structure saying, "I can also conclude that SOP comes out last because that's usually the case I'll put it at the bottom and there's a closing brace at the very end" (in this case the print should come before the increment in the loop).

Even from students who performed well on the interview questions, we still hear the importance of ritualistic knowledge about programming. P04 says "Declarations all go first, I forget if I don't put them all at the front. Answers go last."

Problem solving (in code) like someone else is hard. This comment was ubiquitous – but especially notable among the stronger-performing students. P09 reflects the comments of many stating "[I'd] rather do it in my own. I think when I look at the problem, I formulate what it would look like and I could say in English what I would do."

P05 elaborates about how reading code (in general and on Parsons problems) is harder than writing code – "this gives you less freedom than writing. That's something I don't like. You have to read and understand the code." He goes on to clarify that writing allows you to create with what you already know (using familiar logic and syntax). When questioned, he clarified that the "things he already knew" were in contrast to reading which may require knowledge he perhaps doesn't know or knows fragilely. For example P05 explains "I like to use 0 and i equal together – I don't like i = 1".

Code tracing may not illuminate students' mental models. Though we readily admit the evidence pool is small, of the seven students who traced code in interview, only two did a passable job. Most critically, one student (P11) refused to trace the code. He just said "I don't want to do this, do I have to?" When asked why he says "it would take a long time and be a lot of work." In discussion at the end of the interview, he indicated that tracing was not something he considered important in the real world – he would use a debugger. P13 also struggled with tracing – and apparently had in the past. "My bad habit I don't like to write out what I am doing -- in Java that's kind of a mistake. I should write it out each time. [I treat it like a] logic question -- I try to do it in my head…and usually get it wrong [on exams]." So he knows that he uses poor tracing technique and even has a vague idea of how to do better. When asked why he doesn't take his own advice, he states simply – it's "really annoying."

Other students evidenced bad tracing techniques similar to those seen by the Leeds working group [4]. Specifically, students like to trace by putting the values directly in the code (not making a table) "For the variables, if it has a value, I put that value there. When I read through it there's actually a value there." [P09].

4.4 Analysis

From the comments of students we see evidence of following:

- Students used both simplistic, pattern-based, simple structural rules as well as logical problem solving skills to solve Parsons problems. We find that providing a structured outline encourages strategic techniques (pattern engagement) but can allow one to test "harder" problems on an exam as students have some hint as to the structure of the solution. This allows students to show some knowledge even if the complete solution is out of their grasp.
- We find that the cognitive load needs to be managed to keep cognitive effort focussed on programming issues rather than artefacts of the problem itself. This can be done through good problem design (see Section 6.2, recommendations for writing Parsons problems).
- We find that Parsons problems are perceived as difficult because they require students to read code written by others (using syntax and logic that might not be in their personal comfort zone).
- We find that Parsons problems may be a better indicator of "some knowledge" in a conceptual area than code tracing (which has been indicated to be a "subset" knowledge compared to code writing on exams). Code tracing (compared to Parsons or code writing) seems to be considered a "different beast" by students.

These findings assisted us in designing one instance of a Parsons problem for an exam situation. In part to fit with the requirements of the exam for the class and, in part, to enable a good comparison with a code writing question, we gave a problem that, based on our interviews, would be medium to easy. The code was only six lines long, paired options, asked students to write out lines (not letters), and gave no supporting structure.

4.4.1 What Does a Correct Parsons Answer Mean?

Based on interviewer observation, we can supply additional information on how students approach Parsons problems and what possible value they may have in assessment or exam situations.

Since six students were asked to write code before and after doing their Parsons problems, we can state that it is possible to solve a Parsons problem (even correctly) and still not understand at a deep level how the code works. This was evident from two sources. First some students asked if they could write the code "their own way" even after seeing the solution through completing the Parsons problems (and not just one, but 5 variants). From their discussion, it was clear that they were still more comfortable with some (often poorly formed) idea of their own solution, using structures and syntax of their own choosing. Second, when students would (try to) reproduce the code given to them in Parsons they would sometimes get a key issue wrong (like (r+c)%2 == 0), or struggle to get it right, or talk aloud about how they don't really know exactly what that does, but it seems to work. Often, in these cases, when students were solving the Parsons problems, they picked the right line by tracing an iteration or two of the loop, but not getting a full comprehension of what that line was doing.

Seven students were asked to trace code that was similar in structure to the code they were given (before and after the Parsons problems). The most striking thing here was how little code tracing ability seemed to be related to Parsons ability or the level of understanding that the interviewer was able to identify in a student during the Parsons problem solving process. While students who performed quite poorly on Parsons often exhibited the same cluelessness in tracing, it was harder to differentiate them from those who showed significantly more understanding (when doing Parsons). Perhaps they had spent more time training themselves in this task in their CS1 studies. Some of the most confused could trace somewhat. Some who could "solve problems well" in Parsons refused to trace or did so very poorly.

5. STUDY 2: WHAT DO THESE ASSESS? QUANTITATIVE RESULTS

In this section we report on the marks obtained by 74 students in the code writing and code tracing questions and the Parsons problem on The University of Auckland CS1 final exam during summer semester, 2008. We report on the ability of students to successfully select the correct statements for the Parsons problem, and show how the syntax, logic and understanding of the algorithm can be distinguished in a Parsons problem. We also compare Parsons problems with the more traditional code writing and code tracing questions.

5.1 Marks for each question

The code tracing question was marked out of 5, code writing out of 11 and Parsons out of 9. The distribution of marks for these questions is shown in Figures 3, 4, and 5.

Table 4 shows a summary of the marks for each of the three questions. Here we show both the mean, and student performance based on their quartile rank in the class. We used the full exam marks (as awarded at the University of Auckland) to determine student quartile ranking. Overall, students scored lowest in the code tracing question, higher in the code writing question and highest in the Parsons problem.

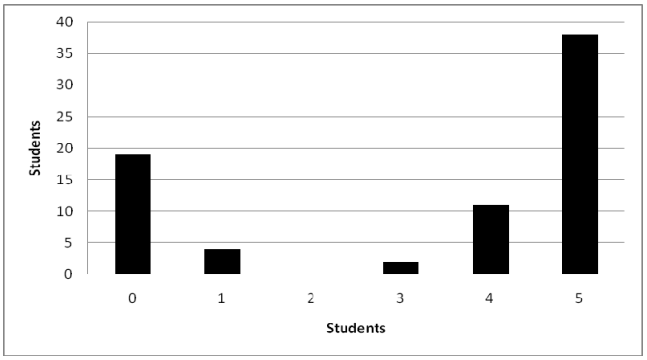

Figure 3. Marks for the code tracing question

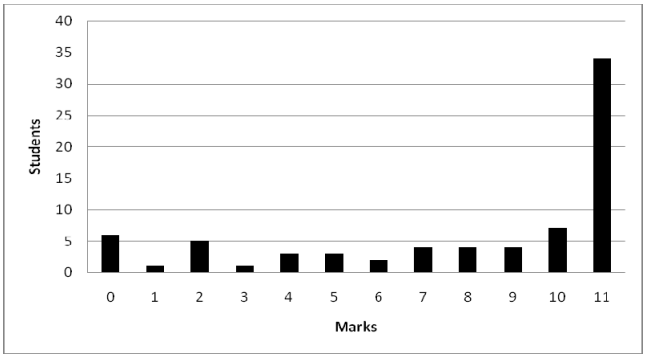

Figure 4. Marks for the code writing question

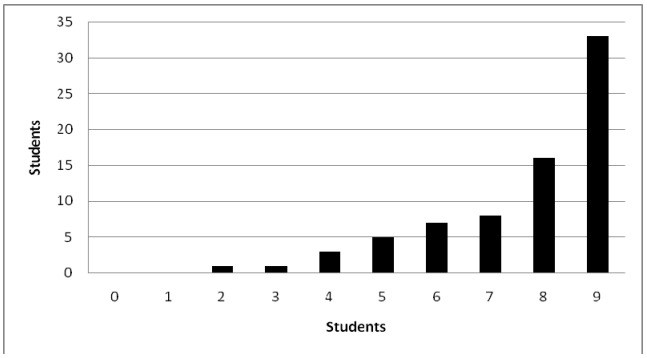

Figure 5. Marks for the Parsons problem

5.2 Relationships between questions

We used the Spearman rank correlation coefficient (because our data are non-normal), r to calculate the r-squared value for the Parsons problem compared to code writing, the Parsons problem

Table 4. Marks for each of the question types

N = 74	Tracing	Writing	Parsons
Lower	0 (0%)	5.25 (48%)	7 (78%)
Median	5 (100%)	10 (91%)	8 (89%)
Upper	5 (100%)	11 (100%)	9 (100%)
Mean	3.25 (65%)	8.01 (73%)	7.64 (85%)
Std Dev	2.19	3.82	1.71

compared to code tracing, and code writing compared to code tracing. The r-squared value can be interpreted as the proportion of the variance in y that is explained by the variance in x. Figures 6, 7 and 8 show the relationships between the three different types of questions, where the area of each circle represents the number of students who achieved the corresponding scores. The Spearman's r-squared for Parsons and code writing is .53, for Parsons and code tracing it is .19 and code writing and code tracing it is .37. We note that the ceiling effect resulting from the large number of students that scored full marks is a significant threat to the validity of the correlations reported here. This is discussed in more detail in section 7.

Overall, student ability as evidenced by performance on Parsons problems is more closely explained by their performance on code writing than it is by code tracing. Additionally performance in code writing is not compellingly explained by code tracing.

Since the relationship between Parsons problems and code writing appears to be the most meaningful, we looked at the difference between the scores obtained for the Parsons problem and the scores obtained for code writing. Figure 9 shows students ranked in descending order according to their full final exam results with the top students to the left and the weakest students to the right. Both the code writing scores and the Parsons problem scores have been scaled to be out of 10 in order to compare them more easily.

The best students obtained full marks for both Parsons and code writing, so the difference is 0. Towards the middle of the graph, we see a number of students who scored lower on the Parsons problem than on the writing question. In the lower quartiles, we can see overwhelmingly that students scored higher in the Parsons problem than in the code writing.

5.3 Syntax and Logic

Table 5 shows the breakdown of results obtained for the Parsons problem. Each pair of statements is categorised as requiring a decision about syntax or logic. The proportion of students that selected the correct statement from each pair is listed.

We identified when students used braces to incorrectly group statements (typically including the return statement within the loop). 85% of students placed the braces correctly.

In addition, we looked at the correctness of the ordering of the statements. A value between 0 and 2 was allocated to the ordering. A score of 2 was assigned to answers that had all the statements listed in the correct order. If one or two statements were out of order, then a score of 1 was assigned. Any answer with more than 2 statements in the wrong place was awarded a score of 0. The results for the ordering of the statements in the Parsons problem are shown in Table 6..

In order to better understand how both strong and weak students perform on Parsons problems compared with code writing questions, we identified students belonging to the upper and lower quartiles when they were ranked according to the final exam score. We took the students in the lower quartile, upper quartile and middle two quartiles, and we calculated the mean score for each of the logic part of the Parsons problem, the logic part of the code writing question, the syntax part of the Parsons problem and the syntax part of the code writing question. The results are shown in Table 7.

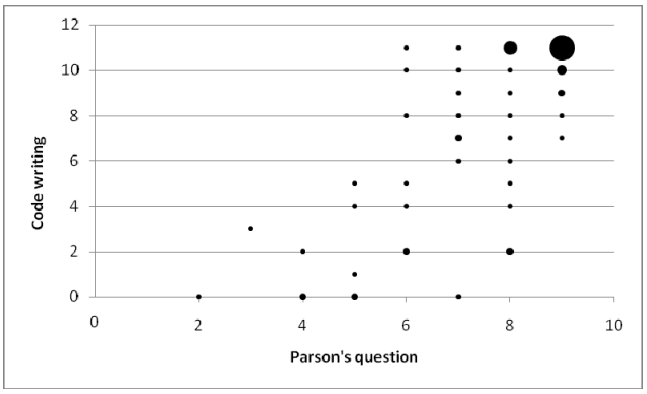

Figure 6. Relationship between Parsons and code writing

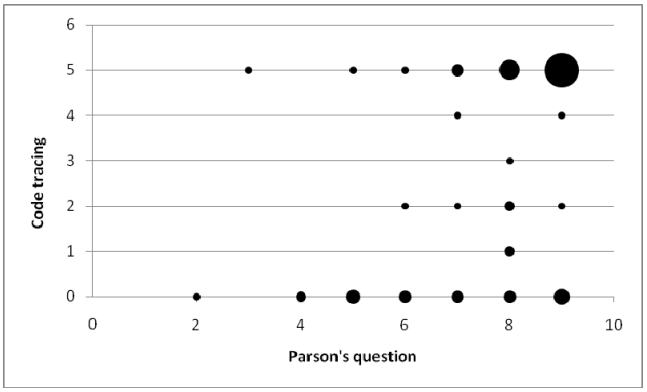

Figure 7. Relationship between Parsons and code tracing

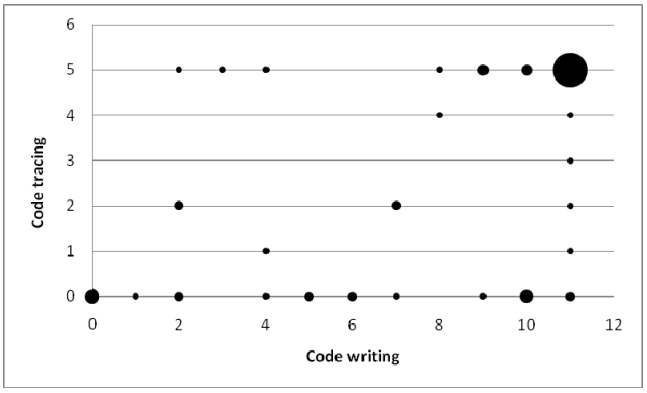

Figure 8. Relationship between code writing and code tracing

5.4 Quantitative Analysis

In this section we analyse the quantitative results.

5.4.1 Marks for each question

The results for the tracing question show a bimodal distribution. This can be explained by the way the question itself was structured. Since the tracing question did not ask for output at different stages in the code, but rather asked for the output after iterating through a loop the required number of times, it was difficult to assign partial marks. Although the marking schedule tried to capture some of the common mistakes that students made and allocate partial marks for understanding some of the code, most of the answers did not include calculations or working which

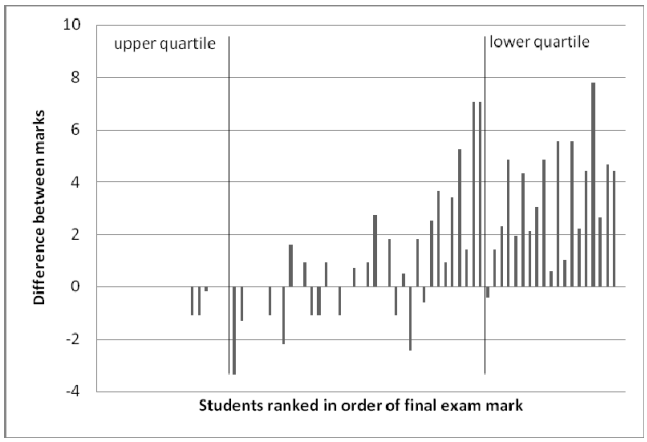

Figure 9. Difference between scaled Parsons and code writing where students are ranked by final exam score.

made it almost impossible to identify how much of the code the student could actually understand. Students that were unable to trace multiple passes over an array typically scored 0.

Both the code writing question and Parsons problem were fairly easy compared to many exam questions, with mean scores of 73% and 85% respectively. It is notable that the upper quartile of all three questions was 100%, which shows that a significant number of the students could accurately answer all three questions.

The lower quartile shows the biggest difference in marks between the different questions. The lower quartile marks were 0% for code tracing, 48% for code writing and 78% for Parsons. Simple guesswork accounts for a portion of the marks in the Parsons problem. A student that has no idea which statement is correct for any given pair still has a 50% chance of guessing the correct statement. On average, a student that knew nothing about programming would guess 3 correct statements (out of the 6) resulting in 3 out of 9 (33.3%). The effect of guesses inflating grades could be limited by multiplying the mark for correct statements by the mark for ordering those statements. The chance of correctly guessing the ordering of statements is sufficiently small as to be of little concern to faculty grading exams for summative assessment purposes. In the future we look forward to evaluating more difficult Parsons problems.

When faced with a challenging writing problem, some students do not know where to start and may leave the page completely blank – six out of our 74 students did. These students often do know something about code, but are not able to show what they know in a writing task. Parsons problems allow students the opportunity to show what they do know. Figure 6 shows that lowest quartile performers scored much higher than the average for guessing (33%) in the Parsons problem. Our six students who scored zero in code writing averaged 46% in Parsons – demonstrating that they do have some understanding.

5.4.2 Relationships between questions

We started this research aware of the possible premise that code tracing is a skill that is cognitively easier than code writing. We anticipated that Parsons problems might act as a middle ground between writing and tracing, since these forms of questions seem to involve the ability to understand individual lines of code, make choices between similar lines of code and arrange those lines of code in a coherent manner that solves the required problem. The r-squared values obtained for the relationships between code tracing, code writing and Parsons did not support our intuitions, but instead suggest that code writing and Parsons problems require the application of similar skills, whereas code tracing requires a quite different set of skills. Performance in code tracing is explained less by Parsons problems than it is by code writing. It appears that selecting and ordering lines of code involves some of the same cognitive problem solving skills as code writing.

We considered that students may have used different approaches to solve Parsons problems than those used for the code tracing and code writing questions. For example, students might be able to arrange the statements in the correct order using strategies such as putting the line containing the header at the start, the return at end, ensuring that variables are declared before they are used, and so on. Knowledge of syntax alone might enable students to obtain a large number of marks without really understanding the algorithm. Although we cannot discount this possibility, the r-squared value for Parsons compared to code-writing is 0.60, which is a powerful indicator that similar skills are involved in code writing and Parsons problems. If students are using syntactic rules to solve Parsons problems without understanding the algorithm, then it is likely that they are using similar strategies to obtain marks for the code writing problems (e.g. identifying the correct formal parameters and return types).

Table 5. Breakdown of the Parsons problem results

Statement pairs	Syntax or logic error	Proportion of students selecting the correct statement
Return	Logic	0.97
Header	Syntax	0.96
Initialisation	Syntax	0.91
Conditional	Logic	0.81
Accumulator	Logic	0.80
Loop	Syntax	0.65

Table 6. Marks allocated for ordering the statements correctly.

Marks	Proportion of students
Correct (2)	0.72
Partially correct (1)	0.26
Incorrect (0)	0.03

Table 7. The mean marks for students in the lower quartile, middle half and upper quartile.

	Logic		Syntax	
	Parsons	Writing	Parsons	Writing
Lower	63%	27%	58%	23%
Middle	91%	86%	91%	77%
Upper	98%	100%	98%	98%

5.4.3 Syntax and Logic

Since Parsons problems consist of lines of code arranged in pairs, and students are asked to select the correct line of each pair to use in the solution, we can analyse the kinds of errors that students make much more easily than with traditional code writing questions. Since the correct option is always visible to students, when they choose the incorrect option we know that it was not a minor typo, but rather that students considered both possibilities and deliberately chose the wrong option. This gives us insight into the kinds of mistakes and misconceptions that students have.

Parsons problems help instructors to identify things that students are struggling with. For example, Table 5 shows that 80% of students chose the correct line to accumulate values. When presented with both alternatives, 20% chose the simple assignment statement rather than the correct statement. This might lead an instructor to consider spending time in class discussing the different roles that a variable might play in a program, and explicitly distinguishing between an assignment operation and an accumulation of a value in a variable. After this intervention, an instructor could use a Parsons problem to again isolate this knowledge during the evaluation of the intervention. The use of Parsons problems allows us to test knowledge needed for a code writing context in a way we can isolate and identify specific misconceptions much more easily.

6. DISCUSSION

6.1 Value for CS1 Assessment

Parsons problems appear to effectively assess similar skills that traditional code-writing questions assess, and at the same time are considerably easier and less subjective to mark. The contrast between Tables 1 and 2, which summarise the marking rubrics we adopted for the Parsons and code-writing problems respectively, is striking. Despite trying to develop an objective marking scheme for the code writing problem, our final rubric still required errors to be classified as either "minor" or "significant" which unsurprisingly lead to a number of discrepancies in the marks assigned by each author.

Analysis of the common errors that students make when answering assessment questions allows instructors to identify concepts with which students are struggling. The open-ended nature of code-writing questions makes identifying such errors difficult, whereas student responses to Parsons problems are well suited to such an analysis.

It is true that for weak students the Parsons problem format provides an opportunity to guess an answer that is not available in code-writing questions. This effect can be ameliorated by asking more than one Parsons problem which requires similar knowledge (but is not detectable as the same at the surface level). Additionally, in interviews we observed that students who were not able to begin answering a code writing question performed much better in the Parsons problem than is consistent with pure guess work. This indicates that the Parsons format provides opportunities for students to demonstrate what they do know that is not available to them in code writing questions.

We believe these reasons offer a compelling argument for instructors to include Parsons problems as part of the formal assessment for their courses.

6.2 Recommendations for Writing and Marking Parsons Problems

Developing a Parsons problem is fairly straightforward. Starting with a correct solution to the problem, rearrange the lines of code into a random order. Then, for each line of code, provide an incorrect alternative which is similar to the line, yet which isolates a misunderstanding of either code syntax or algorithm logic. For clarity, it is best to highlight the division into pairs either by numbering them or by separating adjacent pairs with a blank line.

In terms of the format for answering a Parsons problem, we suggest providing the students with a blank space into which they can rewrite the lines of code they select. An alternative format, where the lines of code are numbered and only the numbers corresponding to the selected lines are written, was not well received by the students in our interviews. This is understandable as although it may take more time, writing out the lines of code in full allows each line to be viewed in context.

For ease of marking, it is important to ensure there is a unique answer to the question, so that for each pair of lines, one must be included in a correct solution and the other one cannot possibly lead to a correct solution.

Developing a marking rubric for a Parsons problem can be as simple as deducting 1 mark for each missing line of code, and deducting 1 mark for each incorrect line of code selected. Another mark can be deducted for braces and/or indenting.

Next, the ordering of the lines needs to be checked, and for this we recommend deducting 1 mark if a single line is out of place or if two lines are transposed and deducting 2 marks for any more serious ordering errors. There are two strategies that can be used to place a greater emphasis on the algorithm logic. Firstly, either more marks can be allocated to ordering, or the ordering mark can act as a multiplier for the marks allocated for selecting the correct lines. In the latter case, it may be reasonable to use 1 as the minimum ordering mark.

Finally, if possible, ask more than one question (tracing, multiple Parsons or code writing) that get at a given concept to be more comprehensive in assessing student understanding.

7. THREATS TO VALIDITY

Students that were interviewed were self-selected and probably more confident than the general student body. They were all North American. A number of them reported that they had some programming exposure (Pascal, Java, C, html) in high school.

The quantitative results are obtained from a CS1 exam conducted during summer school in New Zealand. The summer semester is only 6 weeks long compared to a normal 12 week semester. Students that enrol in the summer semester CS1 tend to achieve higher grades than students in a normal semester. We did not record the time that the students took to complete the exam, so we do not know if they ran out of time.

The code that students were given for the code tracing question had some notable challenges. The subtraction required to calculate the index of the array appeared to confuse many students. For those experiencing mental or stress loads in the exam, a calculation used in the array index may have looked

daunting. This seems especially likely since the students often didn't look at all 3 lines in the loop. They seemed to focus on the one with the difficult indexing and ignored the line before it which assigned to temp and the line after which assigned temp into the swap location. Although all three questions require iteration over an array, the additional complexity of the code tracing question means that it may not have used the same concept skill set as the Parsons and writing questions.

A significant threat to the validity of our statistical results lies in the fact that a very large number of students scored perfectly on all three exam questions. This sort of score breakdown is often an issue in studies involving actual exam questions. Further studies with more difficult questions (and preferably questions of closer difficulty level) will be important to document well any statistical relationship between Parsons problems and code writing. We strongly urge educators to follow our recommendations for Parsons problem development and report on results from their own exam experience.

8. FUTURE WORK

We would like to continue our work with Parsons problems by examining different variants and enhancements which would allow us to make more accurate conclusions about what students understand.

One obvious variation is to use more than two alternatives for each line of code. This would reduce the effect of guessing on student results. Assessing the same concept on multiple Parsons would also address this – what percentage of students would answer consistently across problems? Another example, suggested by a student in the interviews, is to require an explanation for why the line that was not selected was incorrect. This may eliminate guesswork in the selection of lines at significant marking cost.

We would also like to explore a range of difficulty levels for Parsons problems. It seems reasonable that students would be able to generate final code for a Parsons problem which is more complex than what would be expected in a code writing question. We are also considering applicability outside of CS1.

We suspect that students can complete a Parsons problem faster than they could complete an equivalent code writing task, but we have no timing data as yet. We would like to collect data about the time it takes to complete these tasks and look at the effects of using multiple Parsons problems in a single exam to further explore the measurement of student understanding.

9. CONCLUSIONS

We have performed an initial investigation of Parsons problems for a CS1 exam setting through interviews and quantitative exam question comparisons. Parsons problems are easier and more reliable to mark than code writing, provide an opportunity to test student misconceptions more specifically than code writing, yet they appear to require the same set of skills (as analyzed by correlation in marks achieved). This makes them an excellent alternative to traditional code writing questions, and we encourage others to use these kinds of questions in their exams and to explore the potential of Parsons problems further.

10. ACKNOWLEDGEMENTS

This work is part of the BRACElet project and was funded in part by a SIGCSE grant.

11. REFERENCES

[1] Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E. and Whalley, J. The teaching of novice computer programmers: bringing the scholarly-research approach to Australia. In *Proc. Of ACE 2008, Wollongong, NSW, Australia*. CRPIT, 78. Simon and Hamilton, M., Eds., ACS. 63-68, 2008.

[2] Garner, S., The Learning of Plans in Programming: A Program Completion Approach. In *Proc. of ICCE'02*. IEEE Computer Society, Washington, DC, 2002.

[3] Hieggelke, C., Maloney, D., Kanim, S., O'Kuma, T. E&M TIPERs: Electricity and Magnetism Tasks Inspired by Physics Education Research. Pearson, Prentice Hall, 2006.

[4] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Mostrom, J. E., Sanders, K., Seppala, O., Simon, B., & Thomas, L. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin*, 36(4): 119-150, 2004.

[5] Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy, In *Proc. of ITICSE 2006*, Bologna, Italy. 118-122, 2006.

[6] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. A Multi-Institutional, Multi-National Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bulletin*, 33(4).125-140, 2001.

[7] O'Kuma, T., Maloney, D., Hieggelke, C. Ranking Task Exercises in Physics. Prentice Hall, 2000.

[8] Parsons, D. and Haden, P. Parsons' programming puzzles: a fun and effective learning tool for first programming courses. In *Proc. of ACE 2006*, Hobart, Australia, 157-163, January 16 – 19, 2006.

[9] Selden, J. and Selden, A. Unpacking the logic of mathematical statements. *Educational Studies in Mathematics*, 29(2), 123-151, 1995.

[10] Selden, A. & Selden J. Validations of proofs considered as texts: Can undergraduates tell whether an argument proves a theorem? *Journal for Research in Mathematics Education 34*, 1, (pp.4-36), 2003.

[11] Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M. and Robbins, P. Bloom's Taxonomy for CS assessment. In *Proc. of ACE 2008*, Wollongong, NSW, Australia. 155-162, 2008.

[12] Whalley, J., Clear, T. & Lister, R. The Many Ways of the BRACElet Project. *Bull. of Applied Computing and Information Technology* Vol. 5, Issue 1. ISSN 1176-4120, June 2007. Retrieved April 24, 2008 from http://www.naccq.co.nz/bacit/0501/2007Whalley_BRACELET_Ways.htm