Stepping Up to Integrative Questions on CS1 Exams

Daniel Zingaro
Ontario Institute for Studies in Education
University of Toronto
daniel.zingaro@utoronto.ca

Andrew Petersen
Dept. of Mathematical and Computational Sciences
University of Toronto Mississauga
andrew.petersen@utoronto.ca

Michelle Craig
Dept. of Computer Science
University of Toronto
mcraig@cs.toronto.edu

ABSTRACT

In this paper, we explore the use of sequences of small code writing questions ("concept questions") designed to incrementally evaluate single programming concepts. We report on a study of student performance on a CS1 final examination that included a traditional code-writing question and four intentionally corresponding concept questions. We find that the concept questions are significant predictors of performance on both the corresponding code-writing question and the final exam as a whole. We argue that concept questions provide more accurate formative feedback and simplify marking by reducing the number of variants that must be considered. An analysis of responses categorized by the students' previous programming experience suggests that inexperienced students have the most to gain from the use of concept questions.

Categories and Subject Descriptors

K.3.2 [**Computer Science Education**]: Computer and Information Science Education

General Terms

Measurement

Keywords

novice programming, exams, CS1

1. INTRODUCTION

Learning how to program requires understanding a challenging set of concepts and skills, each of which requires significant time and effort to develop. In an introductory programming course, each student's programming abilities develop gradually over the term, and as these abilities become more robust, the student is able to undertake and complete increasingly complex tasks. The ability to handle these tasks is related to self-efficacy, so providing structured, accurate feedback and acknowledging and rewarding progress are crucial. However, assessments in introductory programming often require students to complete large pieces of code, and the danger is that students who cannot successfully complete a program will be unable to identify how much they do know and where they need to improve.

The way in which a student represents a problem has a direct bearing on the problem's difficulty for that student [2]. Both novices and experts begin solving a problem by deciding to which "category" the problem belongs, then use knowledge associated with that category to form a solution. Unfortunately, novices often categorize by using surface features of the problem, rather than an understanding of the underlying principles or core concepts. In a physics study, novices (first-semester mechanics students) grouped conceptually-unrelated problems based on similar objects or physics terms in the problem text [2]. A corresponding study in computing [12] found similar results: when given a list of 27 problem specifications and asked to sort them based on solution type, novices grouped them by application area (e.g., business, operating systems), whereas experts grouped them on the basis of the algorithm required in their solution.

More recently, it has been argued that students have difficulty seeing the "big picture" of a problem, tending instead to focus on individual components or line-by-line explanations [5, 10]. According to the SOLO taxonomy [1], such students are working at a multistructural level rather than a relational level. For example, students are often unable to explain the purpose of a small piece of code, even when presented in multiple choice format [13].

That students often evoke suboptimal problem representations and fail to abstract from the given problem statement suggests a particular difficulty with large, integrative code-writing questions on tests and exams. Due to the high proportion of such questions, our final exams may not afford students a reasonable opportunity to demonstrate what they know. Code-writing questions intended to target specific concepts may involve many more concepts than expected, and fragile understanding of any of these concepts leads to poor performance overall [7]. Here, we contend that code-writing questions are difficult for novices because they require the students not only to use many concepts, but also to know which of those concepts to use in the first place.

In this paper, we consider an alternative to large, integrative code-writing questions: sequences of small questions that target individual concepts by asking students to write fragments of code. We argue that such questions allow for more fine-grained measures of student ability, and that they

SIGCSE'12, February 29–March 3, 2012, Raleigh, North Carolina, USA.

can serve as powerful indicators of performance on integrative questions.

2. RELATED WORK

Using published lists of CS concepts [9, 4], Petersen et al. [7] investigated the number of concepts embodied in typical CS1 final exam questions. In a sample of fifteen CS1 final exams, the authors found that 59% of exam marks were devoted to code-writing questions. On average, each such question required students to understand and use four "significant" concepts (excluding, for example, fundamental syntactic and structural concepts). It appears that programming questions are, at least in terms of the number of concepts, more difficult than instructors expect.

In a similar study, Sheard et al. [11] sought to develop a classification scheme with which to categorize CS1 exams on a number of dimensions. The authors found that it was quite difficult to agree on whether a question was easy, medium, or hard. Furthermore, the authors tended to identify large numbers of concepts per question; they provide an anecdote describing the identification of fourteen topics for a single multiple choice question. These results therefore show that other question styles can also test students simultaneously on many topics.

As CS1 teachers, our goal is to help students ultimately write code. Yet, code-writing has been found to be the most difficult of a hierarchy of tasks that includes code reading and tracing, explaining, and writing [15]. Researchers have therefore investigated the utility of these other types of questions in terms of predicting performance on code-writing questions. For example, Lopez et al. [6] found in a final exam that performance on two types of questions — tracing code containing a loop, and explaining code — accounted for 46% of the variance exhibited in the code-writing portion. In a similar study, Venables et al. found that "explain" questions accounted for 49% of the variance on the writing questions, but that much of this predictive power is restricted to students who received maximum points on the "explain" questions. When these students were removed, only 6% of variance was explained [14]. Tracing questions accounted for a similar percentage of variance, but in contrast to the "explain" questions, the linear relationship applies to students who fall *below* a certain threshold on the tracing questions. That is, those who do poorly on tracing do poorly on code-writing, but doing well on tracing does not imply doing well on code-writing.

Parsons problems have also been suggested as a tool to assess requisite code-writing knowledge. In a Parsons problem, students are provided with a random permutation of the lines of code required to solve a problem and must reorder that code correctly. Often, a superset of the code lines is provided, where each line must be selected from a pair of similar lines. Research has found that performance on Parsons problems is highly correlated with code-writing but not with code-tracing [3]. In addition, inter-rater reliability on marking a Parsons problem was substantially higher than agreement when marking a code-writing question. Rubrics for Parsons problems require fewer cases than code-writing rubrics and have better-defined marking criteria.

The ease and accuracy by which Parsons puzzles can be marked suggests that such puzzles can serve as a source of consistent feedback for students. Yet, there is evidence that solving a Parsons puzzle does not necessarily mean that students can write code. In a qualitative interview study, students were given a Parsons puzzle and, once they solved it correctly, were asked to write that solution code from scratch. Many students were unable to do so, suggesting that heuristic or contextual cues can be used to solve a Parsons puzzle without fully understanding the solution [3]. In addition, since the code is already "on the page", students are not themselves writing any code.

In this paper, we are interested in the use of focused code-writing tasks where students write small amounts of code. Our goal is to be able to test concepts in relative isolation, limiting larger design or problem-solving requirements. We are also interested in providing consistent and accurate formative feedback to students about their mastery of individual concepts.

3. STUDY CONTEXT

In the fall of 2010, we conducted a study to investigate the utility and appropriateness of small code-writing questions on CS1 written examinations. Our goal was to test understanding of single concepts (such as being able to loop over a list, use an if-statement, or use variable assignment) with very little distraction or dependence on other concepts. However, topics in CS1 are graduated in the sense that it is hard to imagine being able to write code that uses an accumulator without first understanding a looping structure. Therefore, when we could not target what we considered to be an important concept directly, we created several graduated questions leading up to that concept. Each such question contains a short statement of the problem and typically requires one to five lines of code. We refer to these questions as **concept questions** and, in some cases, labeled them with the concept being evaluated.

Our full study evaluated responses on term test and final exam questions from four CS1 courses with a total of 642 consenting undergraduate students. Each student provided information about their previous programming experience and their intentions to major in CS. The exams and tests included concept questions and standard code-writing questions, as well as a mix of code-explaining and tracing questions.

This paper reports on five questions (one code-writing and four concept questions) on one final exam. The concept questions were intentionally written to match the concepts tested in the code-writing question. In addition, the students had encountered similarly-styled concept questions on a test given earlier in the term, which should reduce any effect caused by unfamiliarity with a new type of question.

The exam was graded out of 73 marks, including 11 marks for concept questions, 22 marks for tracing, 30 marks for code-writing, 3 marks for code-explaining, and 6 marks for sorting algorithms. Of the 193 students initially registered in the course, 112 (58%) provided consent to include their data in the study. A total of 123 students completed the term work and final exam; of these, 77 (63%) provided consent. Our consenting subset averaged 70% on the exam; the entire class averaged slightly lower at 67%. All questions on the exam were graded by the course instructor.

3.1 The Questions

The code-writing question we studied and one possible solution are presented in Figure 1. We refer to this question as

A list of students and their assignment marks has the following format. Each element of the list is a tuple where the first element is the student ID and the second element is a list of marks. Here is an example:

```
marks = [('dan11', [76,80,67]), ('jane23', [81,90,69]),
         ('jones11', [77,79,55])]
```

In the list of assignment marks, element 0 corresponds to assignment 0. Every assignment is out of 100 marks. Complete `calc_average` so that it works according to the docstring.

```
def calc_average(a_num, marks):
    '''Return the average mark on assignment a_num for
    all students in list marks.
    Precondition: marks contains an assignment
    corresponding to a_num.'''
```

```
    total = 0.0
    for student in marks:
        total += student[1][a_num]
    return total / len(marks)
```

Figure 1: The code-writing (*write*) question with solution entered into the answer box.

write. While this question does not require much code (not much more than the concept questions introduced later), students must understand several concepts and, as importantly, identify the need for specific programming structures and integrate them. They must use a loop to iterate over the tuples in the outer list, be able to use list-indexing, and understand the idea of nesting inner lists within the tuples. They must maintain an accumulator inside their loop (i.e., by using an expression) and, following the loop, divide that accumulator by the length of the list (another expression). Though the function header is given, elements of function structure remain pertinent: students must include a `return` statement and understand that the parameters are accessible as variables within the function.

This question was marked using an additive scheme worth a total of five marks. Students earned one mark for each of looping through the list, initializing and updating the accumulator, indexing a tuple's second element (dealing with the nested structure), and accessing the correct assignment in the inner list. They earned 0.5 marks each for a `return` statement and dividing the accumulator by the number of marks. Syntax issues were generally ignored.

To target isolated elements of **write**, we created four concept questions worth one mark each. They appear in sequence in Figure 2 with sample correct solutions provided in the answer boxes. The bold font labels describing what is required in each question (e.g., **Accessing a List Element**) were included on the examination paper.

The first of these concept questions asks whether students can index a list in isolation, and the second determines whether students can perform a basic loop over a list. We anticipated that this would be the combined minimum knowledge required to make any progress whatsoever on **write**. The third concept question adds one level of nesting to a list, and the fourth introduces the idea of an accumulator. Together, the latter two questions capture what we think of

Accessing a List Element. Write a single Python statement to follow the code below that prints the value `'corn'` by accessing the third element from the list `food`.

```
food = ['peas', 'carrots', 'corn', 'potatoes']
```

```
print food[2]
```

Iterating Over a List. Assume you have the Python variable `friends` that refers to a list of strings. Write Python code to print each element of the list on a separate line.

```
for friend in friends:
    print friend
```

Nested Lists. Python list `L` is a list of lists that could look as shown below. Using a `for` loop, extract and print the second element of each of the sublists of `L`, each on its own line. For example, if `L` were set as follows:

```
L = [ ['a', 'this_one', 'b'], ['c', 'that_one', 'd'] ]
```

your `for` loop would print:

```
this_one
that_one
```

```
for sublist in L:
    print sublist[1]
```

Accumulating from a List. Assume you have the Python variable `prices` that refers to a list of floating point numbers. Write Python code to print the total of all the elements in `prices` without using the built-in `sum` function.

```
total = 0.0
for price in prices:
    total += price
print total
```

Figure 2: The *access*, *iterate*, *nest*, and *accumulate* concept questions. Solutions have been entered into the answer boxes.

as the core of **write**. For the remainder of the paper, we refer to these concept questions as **access**, **iterate**, **nest**, and **accumulate**, respectively.

Most concept questions were marked as 0 (incorrect) or 1 (correct). On **accumulate**, 12 students received 0.5 for either omitting the final print statement, not initializing the accumulator, or including extraneous code such as an unnecessary type cast.

4. RESULTS

4.1 Performance by Quartile

Table 1 presents student performance for each of the five questions of interest. The average for the class and the average for each quartile is presented. The quartiles are based on total exam mark.

The overall average on the final exam was 70%. The questions we studied covered the most basic material on the exam, so they each have substantially higher averages

	write	**access**	**iterate**	**nest**	**accumulate**
Mean	4.09 (82%)	0.95	0.95	0.79	0.90
Q1	2.69 (54%)	0.94	0.81	0.39	0.72
Q2	4.39 (88%)	0.875	1.0	0.8	0.93
Q3	4.34 (87%)	1.0	1.0	0.95	0.92
Q4	4.80 (96%)	1.0	1.0	1.0	1.0

Table 1: Average performance on each question for the class as a whole (Mean) and for each quartile as determined by the total exam mark.

than this. Unsurprisingly, the average for **write** was lower than almost all of the concept questions. **nest** had a slightly lower average, suggesting that the concept of nesting is particularly difficult.

The quartile data gives some indication of the types of problems experienced by various achievement groups in the class. For example, the students in the lowest quartile (Q1) perform very poorly on **write**, but perform well on **access** and **iterate**. It is likely that, for these students, a combination of the larger context of **write** and trouble with nesting and accumulating leads to poor performance on the larger code-writing task. However, the second quartile was able to perform well on **write** despite relatively lower performance on **nest**. This suggests that despite being at the core of the question, nesting is not well represented in the final mark for the question.

4.2 Performance of Subgroups

As explained earlier, we experienced difficulty in targeting concepts that seemed to have necessary prerequisites. In **accumulate**, for example, we additionally tested their ability to write a loop over the list. One can imagine an alternative, where we provide students the skeleton of a loop and ask them to write the one-line accumulator assignment inside. Doing so would likely "give away" **iterate**.

We believe that our concept questions, therefore, are hierarchical, in the sense that **nest** and **accumulate** require the use of the concepts evaluated in **access** and **iterate**. We expect this claim to be borne out in student performance on **write**.

First, we expect that students who cannot answer **iterate** will fail to make much progress on **write**. Only three of the 77 students answered **iterate** incorrectly; on **write**, they obtained scores of 2.5, 2.5, and 0. Based on the marking rubric, these students received some marks on **write** by completing non-central aspects of the question like initializing a variable that could be used as an accumulator and by including a `return` statement.

Turning to **accumulate**, 2 students scored 0, 12 students scored 0.5, and 63 students scored 1. Students could obtain a 0.5 for making a small syntax mistake. On **nest**, 16 students scored 0 and 61 scored 1. The performance of these groups on the **write** question is broken down in Table 2. Among those students who answered both **nest** and **accumulate** correctly, the average mark on **write** was 4.44. **Nest** and **accumulate**, therefore, appear to be a prerequisite for **write** but are not wholly sufficient. That is, **write** includes "more" than the sum of its major concepts. The relationship between **nest** and **accumulate** is less clear; one is not necessarily a prerequisite for the other, though **nest** in general appears to be more difficult.

accumulate Score (# students)	0 (2)		0.5 (12)		1 (63)	
nest Score (# students)	0 (2)	1 (0)	0 (6)	1 (6)	0 (8)	1 (55)
Mean **write** Score	1.25	N/A	2.33	4.33	3.53	4.44

Table 2: Performance on *nest* and *accumulate* is indicative of performance on *write*. *nest* is the more difficult of the two questions.

4.3 Predicting Performance

Writing code is the major determinant of whether a student passes CS1, and the widespread use of integrative code-writing questions [7, 11] suggests that CS1 instructors believe they are important assessments of student understanding. We are therefore interested in measuring the extent to which concept questions measure what is measured by writing questions.

To address this, we performed a multiple regression with **iterate**, **nest**, and **accumulate** as independent variables, and **write** as the dependent variable. (**access** was dropped from this and all future regressions since it did not contribute to predictive power.) A power transform on **write** was required to create normally-distributed residuals. Assumptions of homoscedasticity (non-significant Breusch-Pagan test) and imperfect multicolinearity (VIF = 1.29) were tenable.

The three concept questions explained approximately 37% of the variance in the code-writing question ($F = 14.2$, $p = 0$, R-squared = 0.37, adjusted R-squared = 0.34); each of **iterate** and **nest** on its own was a significant predictor (**accumulate** was weakly significant). A nonparametric Fisher's exact test of independence, which is resistant to small sample sizes, confirms that scores on the concept questions (all correct vs. at least one mark lost) and **write** (80% and greater vs. below 80%) are significantly dependent ($p = 0$). This relationship is very similar to results from Venables et al. [14] reported earlier. That is, when targeting the key concepts of a code-writing question, it appears that our concept questions, like tracing and explaining questions, significantly relate to code-writing questions.

Since the three concept questions were selected because of their connection to the concepts evaluated by **write**, this strong relationship is perhaps unsurprising. However, the remainder of the final exam tested other concepts, many of which we perceived to be more complex than the ones tested by **write**. We therefore ran a multiple regression using the same independent variables but with the final exam mark (minus the contributions of the three concept questions) as the dependent variable. Here, performance on the three concept questions accounts for about 47% of variance on the rest of the final exam mark ($F = 21.51$, $p = 0$, R-squared = 0.47, adjusted R-squared = 0.45). In addition, we find that **write** is only slightly more predictive on the remainder of the final exam mark than are the concept questions ($F = 75.27$, $p = 0$, R-squared = 0.50, adjusted R-squared = 0.49). Together, these results suggest that our concept questions capture a component of ability that is also being measured by the rest of the exam.

Many students answered all three concept questions cor-

rectly, and correspondingly did very well on **write** (on average, 4.47 out of 5). To investigate whether our relationships hold with weaker students, we dropped all students who scored a perfect three out of three on the concept questions, and ran a multiple regression of the remaining concept question scores against **write**. We find a significant linear relationship ($F = 5.39$, $p = 0.007$, R-squared = 0.46, adjusted R-squared = 0.37). This is in contrast to other published comparisons of question types where a linear relationship to code-writing is nonexistent when the high-performers are removed [14]. This may be due to our concept questions requiring code-writing skills, even if they don't require the ability to compose multiple structures.

As a final analysis, we were interested in the extent to which the three concept questions predicted performance on the three other code-writing questions on the exam. The first such code-writing question used the same setup as **write**, but asked students to write a function to add one new mark for each student. Unfortunately, scores on this question were extremely negatively skewed, and normality of residuals could not be attained. The second question asked students to apply a list of weights to assignment marks in order to return a new tuple mapping students to overall assignment grade. The linear relationship was weak (accounting for 20% of the variance); only **nest** was a significant predictor. Finally, the third question required the use of dictionaries to keep a count of the number of each character existing in a supplied string. The linear relationship accounted for 41% of the variance, this time with both **iterate** and **nest** as significant predictors. It appears that our concept questions can predict performance on code-writing questions built to evaluate different, unrelated concepts. Some of the necessary concepts, like iteration, are shared by these other code-writing questions, but neither nesting nor the use of an accumulator was required. The nature of this relationship requires future work.

4.4 Discussion

Our concept questions were significant predictors of both **write** and final exam performance as a whole. In addition, **write** predicted final exam performance only slightly better than our concept questions. Based on these results, we suggest two hypotheses.

First, like explanation or tracing questions, concept questions correlate well with code-writing performance [14]. On the surface, this is perhaps unsurprising, as our concept questions do require some code to be written, as compared to explain and trace questions. Yet, the amount of code written for the concept questions is so small that one could argue that it hardly reflects a "genuine" exam code-writing question, or that such concept questions are so easy as to fail as predictors of anything. Instead, we have found that performance on concept questions tells us much about the performance on more integrative code-writing questions.

Second, concept questions are testing only a subset of what the code-writing questions are testing. Since the major concepts required by **write** are embodied in the concept questions, we suggest that any difference in performance between concept questions and **write** is due to a synthesis or design component. In **write**, students are not only required to understand concepts, but to know which of those concepts to use in the first place, and how to synthesize them in a coherent way. Poor performance on **write** is therefore ascribable to fragile concept knowledge, a failure at the design stage, or a combination of the two. Concept questions help us make progress toward disentangling these sources of difficulty.

In this way, we suspect, but have not tested, that concept questions will be more useful than integrative questions in terms of providing formative feedback to students. There is some wizardry involved in arriving at a student's mark using a typical code-writing rubric [3]. Students can sometimes earn a majority of marks on integrative questions through correct "boilerplate" code (like setting up accumulators and return statements) that is nevertheless distant from the question's intended target. Rubrics for concept questions, on the other hand, can be as simple as awarding a mark for a correct answer, or possibly half a mark if the question is readily separable into two independent pieces.

Even if students are given the marking scheme for a typical question, we suspect that they will struggle to determine exactly where marks were lost or what can be done to increase their performance on the next assessment. Our concept questions are labeled with the very concept that they test. Getting a 0 on a concept question therefore carries with it both the realization that they have answered incorrectly, and a cue to the area on which the student should focus.

5. IMPACT OF EXPERIENCE

In a study on the relationship between self-efficacy and mental models and learning to program, Ramlingam et al.[8] demonstrate that self-efficacy is influenced by previous programming experience and in turn affects course performance. These authors claim that instructors of introductory programming can increase student performance through activities that directly affect self-efficacy. They advocate that instructors "must challenge students but not overwhelm them with complex programming tasks that undermine their self-efficacy" and further suggest that "students need to incrementally build up a history of success at increasingly difficult tasks."

We postulated that the large integrative code-writing tasks may be overwhelming for beginning programmers and that the use of independent concept questions would help students to demonstrate success on the pieces of content that they had mastered. To the extent that this is true, concept questions may serve as affirmations of progress.

While all students in our study were learning to program, students arrive in their first programming course with a spectrum of previous experience. We wondered if the ability to integrate the individual concepts was a skill that developed only after a certain amount of time or exposure to writing code. Perhaps the differences between the performance on independent concept questions and integrative questions would depend on the student's level of programming experience. To study this, we included the question, "Before this course, have you had any experience programming in any computer language?" on the survey accompanying the participant consent form.

The 46 experienced students in our consenting group did very well on both **write** and on the concept questions. They averaged 4.39 on **write**, and between 90% and 98% on each concept question. A regression line for these students confirms that those who do well on concept questions do well on code-writing ($F = 6.79$, $p = 0.0008$, R-squared = 0.33, adjusted R-squared = 0.28).

A regression line explains a similar amount of variance for the inexperienced subset. Yet, these inexperienced students did correspondingly more poorly, averaging 3.64 on **write**, 0.90 on **access**, 0.97 on **iterate**, but only 0.61 on **nest** and 0.84 on **accumulate**. We hypothesize that this increased variance in concept scores supports the utility of these questions as diagnostic tools. A Fisher's test of independence is inconclusive, though it weakly confirms that scores on the concept questions (all correct vs. at least one mark lost) and **write** (at least 80% vs. below 80%) are not significantly dependent ($p = 0.06$). The corresponding Fisher's test on the experienced students was highly significant ($p = 0$). This result suggests that, particularly for inexperienced students, code-writing questions are more difficult than what might be expected by conceptual performance alone. We hypothesize that, while some inexperienced students obtain conceptual knowledge, they have not had sufficient time during which to be able to flexibly use that knowledge to solve new problems in an exam situation.

We also hypothesize that our concept questions help us address the call of Ramlingam et al.[8] to allow novices to build confidence based on what they know. Our results indicated that inexperienced students did well on some of the concept questions, even in the face of poor performance on **write**. We believe, in the case of term tests, that such successes can be important for increasing self-efficacy. Even though novices might not be able to write code on a first term test, we do wish to acknowledge what they **have** learned.

6. CONCLUSION

Traditional code-writing exam questions seem to require a mastery of several concepts, plus the ability to design with or synthesize those concepts. Rubrics for such questions are difficult to create and use and, we suggest, equally confusing as a formative feedback mechanism. Students receive marks for peripheral code in addition to code that satisfies the question's purpose; the mark received therefore may not alert students to core misunderstandings. We propose that single-concept questions — questions targeting one concept, or adding one concept over a previous concept question — are more effective formative feedback tools. We have shown that such questions correlate strongly with code-writing questions and the final exam mark, suggesting that what they target is a significant component of what we care to test. That concept questions are less predictive for inexperienced students suggests a disconnect between conceptual knowledge and code-writing ability for these students. Our tentative suggestion is that some (but not all) of the code-writing questions on tests can be replaced by concept questions, more clearly separating concept-understanding from larger problem-solving and design considerations. We of course believe that all of these abilities are important outcomes of CS1, but we assert that questions targeted to each skill are more indicative of specific abilities than questions testing these skills in tandem.

7. REFERENCES

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

[2] M. Chi, P. Feltovich, and R. Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5:121–152, 1981.

[3] P. Denny, A. Luxton-Reilly, and B. Simon. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*, pages 113–124, 2008.

[4] K. Goldman, P. Gross, C. Heeren, G. L. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Setting the scope of concept inventories for introductory computing subjects. *Transactions on Computer Education*, 10(2):1–29, 2010.

[5] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bulletin*, 38(3):118–122, 2006.

[6] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceeding of the Fourth International Workshop on Computing Education Research*, pages 101–112, 2008.

[7] A. Petersen, M. Craig, and D. Zingaro. Reviewing CS1 exam question content. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pages 631–636, 2011.

[8] V. Ramalingam, D. LaBelle, and S. Wiedenbeck. Self-efficacy and mental models in learning to program. *SIGCSE Bulletin*, 36(3):171–175, 2004.

[9] C. Schulte and J. Bennedsen. What do teachers teach in introductory programming? In *Proceedings of the Second International Workshop on Computing Education Research*, pages 17–28, 2006.

[10] J. Sheard, A. Carbone, R. Lister, B. Simon, E. Thompson, and J. L. Whalley. Going solo to assess novice programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, pages 209–213, 2008.

[11] J. Sheard, Simon, A. Carbone, D. Chinn, M.-J. Laakso, T. Clear, M. de Raadt, D. D'Souza, J. Harland, R. Lister, A. Philpott, and G. Warburton. Exploring programming assessment instruments: A classification scheme for examination questions. In *Proceeding of the Seventh International Workshop on Computing Education Research*, pages 33–38, 2011.

[12] J. Shertz and M. Weiser. A study of programming problem representation in novice and expert programmers. In *Proceedings of the Eighteenth Annual Computer Personnel Research Conference*, pages 302–322, 1981.

[13] Simon and S. Snowdon. Explaining program code: Giving students the answer helps – but only just. In *Proceeding of the Seventh International Workshop on Computing Education Research*, pages 93–100, 2011.

[14] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, pages 117–128, 2009.

[15] J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. Kumar, and C. Prasad. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Proceedings of the 8th Australian Conference on Computing Education*, pages 243–252, 2006.