How (not) to write an introductory programming exam

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Judy Sheard
Monash University
Australia
judy.sheard@monash.edu

Daryl D'Souza
RMIT University
Australia
daryl.dsouza@rmit.edu.au

Mike Lopez
Christchurch Polytechnic Inst of Tech
New Zealand
mike.lopez@cpit.ac.nz

Andrew Luxton-Reilly
University of Auckland
New Zealand
a.luxton-riley@auckland.ac.nz

Iwan Handoyo Putro
Monash University
Australia
iwan.putro@monash.edu

Phil Robbins
Auckland University of Technology
New Zealand
phil.robbins@aut.ac.nz

Donna Teague
Queensland University of Technology
Australia
d.teague@qut.edu.au

Jacqueline Whalley
Auckland University of Technology
New Zealand
jacqueline.whalley@aut.ac.nz

Abstract

The computing education literature shows some recent interest in summative assessment in introductory programming, with papers analysing final examinations and other papers proposing small sets of examination questions that might be used in multiple institutions as part of a benchmarking exercise. This paper reports on a project to expand the set of questions suitable for use in benchmarking exercises, and at the same time to identify guidelines for writing good examination questions for introductory programming courses – and, by implication, practices to avoid when writing questions. The paper presents a set of ten questions deemed suitable for use in the exams of multiple courses, and invites readers to use the questions in their own exams. It also presents the guidelines that emerged from the study, in the hope that they will be helpful to computing educators writing exams for their own courses.

Keywords: introductory programming, CS1, assessment, benchmarking, examination.

1 Introduction

McCracken et al (2001) appeared to discover that many of the students who pass programming courses cannot actually program. The BRACElet project (Whalley et al 2006) explored this issue in great depth and effectively confirmed the problem. Addressing the question of how students might be able to pass programming courses without being able to program, Traynor et al (2006) provided some insight with this excerpt from a student interview: "Most of the questions are looking for the same thing, and you usually get the marks for making the answer *look* correct. Like if it's a searching problem, you put down a loop, and you have an array and an *if* statement. That usually gets you the marks . . . Not all of them, but definitely a pass."

One response to this issue is to analyse the final exams in programming courses, to try to establish how they align with the skills and knowledge that students are expected to acquire. Simon et al (2010) analysed data structures exams in this light, and Petersen et al (2011) and Sheard et al (2013) looked at introductory programming exams.

In an early stage of the current project, 11 common questions were included in the introductory programming exams of six institutions in Australia and New Zealand (Sheard et al 2014). We concluded that four of the questions were suitable for benchmarking purposes, and invited other academics to use these questions in their own exams and compare their students' performance with the published results.

Benchmarking is not an attempt to impose uniformity on courses and assessments across the sector. Rather, it is a way of permitting comparisons: does university A, which has a high reputation and a correspondingly high entry requirement, produce better student outcomes than university B, which accepts the students who are not admitted to the other universities?

Such questions cannot be reasonably asked until there is a meaningful way of answering them. This is what we believe to be the purpose of benchmarking. If interested participants at different institutions can include a reasonable set of common questions in their final examinations, they can compare the results of their students with a published benchmark and form their own conclusions as to the quality of their courses in the context of the student cohorts that they attract.

In reducing an original set of 76 questions to the final 11 (Sheard et al 2014), we noted a number of reasons why participants did not consider questions suitable for use across multiple institutions:

- *Question is too easy.*
- *Question is too large.*
- *Topic is too advanced or not usually covered in a typical introductory programming course.*

- *Student may not be familiar with the style of question.*
- *Style of question is not suitable for an exam situation, e.g. is it reasonable to ask students to identify syntax errors?*
- *Wording of the question is unclear or ambiguous.*
- *Question is idiosyncratic, e.g. referring to the coding style guide of a particular course.*
- *Question involves tricky code, which may obfuscate its purpose.*

In the current phase of the project we set out to further explore these and other reasons, while at the same time expanding the set of questions that can be used for benchmarking. We thus addressed the following questions:

- Can we identify some principles of good question design that others can apply when writing their own questions?
- Can we identify some aspects of poor question design that others can try to avoid when writing their own questions?
- Can we identify examination questions that a group of instructors would all be willing to use in their introductory programming exams?

2 Research approach

The 11 questions from the previous phase of the project were supplemented by a further 20 candidate examination questions, sourced from the literature (principally from publications of the BABELnot project (Lister et al 2012)) and from questions that had been used in exams at the lead authors' institutions. Two additional versions of one question were added, so that the same basic question could be considered in three distinct forms.

The two lead authors conducted a workshop in conjunction with ACE 2014, for academics with current or recent involvement in assessing students in an introductory programming course. The remaining seven authors joined the project by attending the workshop.

The bulk of the workshop consisted of discussion of the 33 questions. For each question, participants rated the likelihood that they would use it in an introductory programming exam, on a scale from 1 (would definitely not use it) to 5 (would definitely use it). At the same time they were asked to give reasons for their choices. Members were at liberty to change their ratings during or after the discussion of each question.

Discussion was lively on many questions, and most members did not complete the rating exercise in the course of the meeting. Members therefore completed the exercise individually in their own time, and submitted their full set of ratings and reasons to the project leaders for analysis.

Analysis began by considering the simple average rating of each question, resulting in a ranking of the 33 questions. This was then supplemented by a qualitative analysis of the members' reasons for their ranking decisions, which resulted in some re-ordering of the list. Finally, questions were selected from the high end of the ranked list, but with consideration to question types and subject matter, so that we did not end up with a substantial number of similar questions.

3 Issues for consideration

In this section we list and discuss issues that arose as we discussed the questions, both at the workshop and in the subsequent data presented for analysis. The issues are in no particular order, and are grouped where possible.

3.1 Question preambles and complexity

Sheard et al (2013) propose a number of measures of question complexity, some of which they suggest should be avoided, while others might be considered a necessary part of what is being tested. One of the measures to be minimised is linguistic complexity, the complexity of the language in which the question is expressed. The essence of the message is that if a question can be expressed more simply, it should be. Among other considerations, this is likely to assist students with a weak grasp of English.

Linguistic complexity is typically encountered in the preamble to a question, the part that sets the scene for what the students are actually being asked to do. Consider Q4, one of the 11 questions from Sheard et al (2014).

Q4. A dependent child can be very loosely defined as a person under 18 years of age who does not earn $10,000 or more a year. An expression that would define a dependent child is

(a) `age < 18 && salary < 10000`
(b) `age < 18 || salary < 10000`
(c) `age <= 18 && salary <= 10000`
(d) `age <= 18 || salary <= 10000`

This question might appear to be expressed in reasonably clear and simple terms. However, one participant questioned the use of the phrase 'very loosely': what did this signify, and might it confuse students into believing that the subsequent definition was not the one to be implemented? In response to this question, the preamble was rephrased to begin "If a dependent child is defined as…". Another participant then queried the use of the word "If", preferring the question to start "A dependent child is defined as…". This wording was rejected on the basis that it appears to be stating a factual definition of dependent children, whereas the intent was simply to provide a definition that could be used for the purposes of this particular question.

There was broader agreement with regard to other questions. For example, the participants all agreed that Q12 would be easier to grasp if the four initialisations were simply presented as the first line of the code, rather than appearing after it with a message telling students to assume that they took place before it.

Q12. This question refers to the following code, where the variables *p*, *q*, *r*, and *s* all have integer values:

```
if (p < q) {
  if (q > 4) {
    s = 5;
  } else {
    s = 6;
  }
}
```

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

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

What would be the value in variable *s* after the code is executed?

Q1. It is an odd fact that the more people there are in a group, the less pizza each of them will eat. Using the following code, how many pizzas would you expect 10 people to eat?

```
if people < 5:
  pizzas = people
elif people < 10:
  pizzas = 3 * people / 4
elif people < 15:
  pizzas = 2 * people / 3
else:
  pizzas = people / 2
```

Considerations of linguistic complexity lead to the issue of contextualising questions. Some examiners like to set their questions in some sort of real-world scenario, while others prefer to limit the question to explicit instructions as to what is required of the students. Consider Q1: one participant said of this question that "the first sentence is distracting and not relevant to what the code is asking about"; others expressed similar concerns. One said "if *people* should be initialised to 10, say so explicitly". There appear to be two schools of thought in this regard. One suggests that students should be given instructions solely about what is required, with no superfluous information; the other, that reading and understanding superfluous information is a necessary aspect of problem-solving, and can be legitimately included in programming questions. The participants in this study did not reach consensus on this question.

A related consideration is the explicitness of instruction. Another question mentioned in its preamble that the elements of an array were initialised. One participant wanted students to be told what the initial values were, although this was not relevant to what was subsequently being asked.

Another form of question complexity identified by Sheard et al (2013) is called 'external domain reference'. They noted that some questions refer to subject matter that might not be known to students in an introductory programming course, and they distinguished between cases where such knowledge is integral to the question and cases where it is incidental and can be overlooked. Q19 falls into the latter category, which Sheard et al call medium-level external domain reference. One participant remarked that the "Question requires some real-world knowledge about what payments and balances mean, which may make it difficult for some students". Others presumably felt that the question could be answered even by students lacking that knowledge.

Q19. What is the purpose or outcome of the following piece of code?

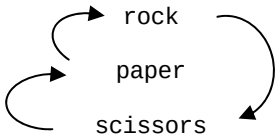
```
for (int i=0; i<payment.Length; i++)
{
  balance = balance + payment[i];
}
```

(a) to add a payment to a balance
(b) to count the payments
(c) to add all payments except the last to the balance
(d) to add all payments to the balance

3.2 Diagrams and examples

In some questions, where it seems that a certain level of complexity is inescapable, diagrams and/or examples can be provided to help students understand the question. Q9 illustrates the point. However, any use of diagrams should be highly contingent on what notation has been used during the course. If students have seen similar diagrams used to explain variable assignment, this diagram would be acceptable; but the final exam is not the place to introduce a graphical notation that the students have not previously encountered.

Q9. There are three integer variables, *rock*, *paper* and *scissors*, which have been initialised. Write code to swap the values in these variables around so that *rock* is given *paper*'s original value, *paper* is given *scissors*'s original value, and *scissors* is given *rock*'s original value. The following diagram illustrates the result of the swaps:



Some participants noted in passing that they were not comfortable with the use of the word 'swap' to indicate movements among more than two items.

When examples are used instead of diagrams or in addition to diagrams, there is a concern that some students will take them as definitive. In Q24, for example, some students might assume that the array will have exactly four elements, and so might write four *if* statements rather than a single *if* statement within an appropriate loop; others might even assume that the code will only be given the array {0, 2, 1, 3}. One participant expressed concern about another question that described an array of unspecified length but gave as an example an array of length 11. But an example is necessarily a particular instance of a generalisation, so it would rarely be possible to provide an example that retains complete generality.

Q24. Suppose you had an array of integers called *mirrors*. Write code that would print out every element of that array that had the same value as its index position. For example, given the array {0, 2, 1, 3}, the code would print the values 0 and 3.

3.3 Material covered in course

It is generally understood that an exam for an early-level course will not test concepts that were not covered in the course. This impacts on our study in that different introductory programming courses do not all cover the same material, even when they are taught using the same language. Questions that are reasonable in the context of one particular course might not be so reasonable in a range of courses at different institutions.

One example of this is the concept of integer division (as in Q1), which one participant describes as "a peculiarity of Java operators being overloaded rather than a core programming concept". It might be reasonable to test the students' knowledge of integer division in a course in which this concept was explicitly taught, but caution should be applied in deciding whether to incorporate the knowledge into questions in other courses.

In addressing our goal of finding a set of questions that can be used in multiple courses using different languages,

we quickly decided that input and output must be regarded as off limits. One obvious reason for this is that different programming languages have very different ways of dealing with input and output. A less obvious reason is that different teaching approaches place different emphasis on input and output. For example, an objects-first approach using Java within the BlueJ environment (bluej.org) need not address I/O at all, as the approach focuses on method calls and their results. Similarly, the media computation approach of Guzdial and Ericson (2013) focuses on the input and output of image and sound files, and touches only briefly on keyboard input, many weeks into the course. A code-tracing question with output statements would therefore be better replaced with an output-less version that asks what values certain variables will have when the code has executed.

Terminology will often differ between courses. Q24, in section 3.2, refers to the 'index position' of an array element. In some courses this might simply be called the index, while in others it might be the position. When adopting questions from other courses, great care must be taken to use the terminology that has been used in the target course.

A further consideration is the preparation that students have undergone during the semester. Some of the questions for our study were provided by a participant who gradually prepares the students for such questions with a series of graded exercises throughout the semester. It seems reasonable to expect that this participant's students would perform better on these questions than students who had not been offered the same preparation.

Finally, consideration should be given to any high-level programming tasks that might be provided by the language being studied, and that might have been covered in the course. Simple array-processing tasks that might be tested in an exam include sorting the elements of an array, reversing the order of elements in an array, and finding the average of the elements in an array of numbers. These tasks become somewhat trivial in a language with inbuilt *sort*, *reverse*, and *average* methods. Even if students have not been taught these features, some might have come across them, and might short-circuit the intention of the question by using them in their answers.

3.4 Variable names (and comments) in code

When code is provided as part of an exam question, the author has three options with regard to the variable names: to make them meaningful, neutral, or 'anti-meaningful' (explained below).

Most programming educators impress on their students the importance of using meaningful variable names, and most apply this practice in their own programming (although many seem not to accept that *temp* and *flag* are sadly lacking in meaning). However, meaningful names can lead students to understand code without having to study the code itself. In Q∞ – which was not part of our study – a student with poorly developed code-reading skills would probably be able to deduce the answer just by reading the variable names.

For examination purposes, therefore, some instructors choose to make the names – or at least those names that might give away the answer – neutral. They might leave *person* and *height* there, to tell students that this is a list or array of people's heights, but replace *totalHeight* and *avgHeight* with, say, *value1* and *value2*.

Q∞. What is the purpose or outcome of the following piece of code?

```
totalHeight = 0
for person in range(0, len(height)):
  totalHeight = totalHeight + height[person]
if totalHeight <> 0:
  avgHeight = totalHeight / len(height)
else:
  avgHeight = 0
```

A number of the code-tracing and code-explaining questions in our study included such neutral names. In one question, the code compares two arrays, returning the last index at which the element in the first array is less than the corresponding element in the second. In a similar question, the code counts the number of times the corresponding elements in the arrays are not equal. Several participants expressed concern that the arrays were called *number1* and *number2*, one suggesting that "it would be better with variable names that provided more meaningful context, for example, arrays of coffee consumed in the morning and the afternoon, and counting the number of days when there are unequal numbers of coffee consumed."

On this same point, consider Q12, in section 3.1. One participant wrote of this question "The responses to the question might be different if the variable names were less abstract and had more context. As academics we often abstract away the variable identifiers as being irrelevant to the question, but then ask students to write code that *does* use meaningful variable names, so our assessment is not well aligned with our expectations of practice. I would use this question with meaningful names." Complying with this expressed need for context might then raise another problem: this particular piece of code might have been written with no real-world context in mind. The variables might simply be numbers, not representing any particular quantities. Should the instructor nevertheless contrive some plausible context? Or is it in fact acceptable to ask students to reason about the code itself, without the additional information provided by meaningful variable names?

Instructors who do use neutral names should consider one further issue: are the different names in the code clear and distinct? During the presentation of a paper at ICER 2013 (Ahadi & Lister 2013) the presenter displayed a code-explaining question and asked why so many students answered it wrongly, and one member of the audience murmured "because they're dyslexic?" The code in the question used two variables, *p* and *q*, which are indeed readily confused by people with certain learning difficulties. The same applies to *b* and *d*. Similarly, the commonly used variable *i* is readily mistaken for the digit 1, which can have a serious impact on a student's understanding of a statement such as *count = count + i*. Instructors who are accustomed to reading and understanding code should take care to ensure that it is not open to misreadings of this sort.

As an aside, most instructors also urge their students to imbue their code with explanatory comments. The code provided for code-tracing and code-explaining questions

tends to have few or no comments, and certainly does not have comments explaining what the code does. Because the code is therefore not of the standard we expect of our students, does this mean that we cannot ask our students to read and explain it?

Finally, in some of our questions the instructors had used what we might call 'anti-meaningful' names, names that have a meaning, but a meaning that appears unrelated to the purpose of the code, and that might therefore mislead students. Instead of a neutral name such as *number1*, an array might be called *fantasy*. Another example is the name *mirrors* in Q24 (section 3.2). The participant who had contributed this question explained that the code was finding array elements that reflect or mirror their indexes. Nevertheless, other participants found the reference a little obscure, suggesting for example that the name *mirrors* might confuse students into thinking about mirror-images of variables, whatever that might mean. In general, it was clear that most of the participants disliked the use of anti-meaningful names.

3.5 Avoidable obfuscation

All computer code has some inherent complexity. However, any task can be coded in different ways that evince different levels of complexity. Is it reasonable to knowingly express the code in a more complex form to test the students' ability to deal with such a form? Q3 provides a simple illustration of this point.

Q3. What will be the value assigned to the variable *x* as a result of the following statement?

```
int x = 10+56 / 5+3 % 12;
```

(a) 13
(b) 11
(c) 24
(d) 10
(e) Generates RunTimeException

The justification for this question was that students had been warned to take care with operator precedence, and that this was a reasonable way to test whether they were doing so. Nevertheless, most participants said that they would use this question only if the spacing were uniform throughout the expression.

Obfuscation can also be unintentional. One example of this is the discontinuity of the code in Q12 (section 3.1); another is the perhaps unthinking use of unnecessary code. In general, participants felt that Q6 tested nothing that would not be tested by a shorter code snippet.

Another question asked students to write a loop to print all the numbers between *p* and *q*, inclusive, that are divisible by *N*. Some code provided to scaffold the question included declarations of *p*, *q*, and *N*, declaration of a scanner, and prompt-input sequences for *p*, *q*, and *N*. The general feeling among participants was that it would be better simply to tell students that the variables had been appropriately initialised, rather than giving them unnecessary input/output code to read.

Q6. What will be printed when the following code is executed?

```
a = 7
b = 3
c = 2
d = 4
e = a
a = b
b = e
e = c
c = d
d = e
print a, b, c, d, e
```

Another form of obfuscation, or tricky code, is code that looks very like something the students have been taught to use and recognise, but with a subtle twist. The last three lines of Q5 look like the standard three-statement swap, but are in the wrong order, and give the same value to each variable.

Q5. What values will the variables *a*, *b*, and *c* have after the following code has been executed?

```
int a = 23;
int b = 11;
int c = 61;
a = b;
c = a;
b = c;
```

We tend to value students who can form an overview of a piece of code without examining it in detail, but this question has the potential to lure these students into a wrong answer, giving the advantage to the struggling but systematic student who needs to work through the code in detail. All of the participants said that they would be willing to use this question, although some proposed that the problem could be overcome by explicitly asking students to trace the code. However, it was considered preferable to test students' tracing abilities with code that is not so easily mistaken for a recognised algorithm.

3.6 A mix of difficulties

Analysing 20 introductory programming exams from ten institutions in five countries, Simon et al (2012) rated the difficulty of every question as low, medium, or high. While three of the exams they studied had no questions of high difficulty, over the 20 exams, nearly a quarter of the questions were rated at the high difficulty level. Examiners clearly believe it appropriate to include a mix of easy, medium, and hard questions in an exam.

Nevertheless, there are some questions in our study that the participants deemed too difficult. One of these was Q2, Soloway's rainfall problem (Soloway 1986), in what appears to be close to its original formulation.

Q2. Read in integers that represent daily rainfall, and print out the average daily rainfall; if the input value of rainfall is less than zero, prompt the user for a new rainfall.

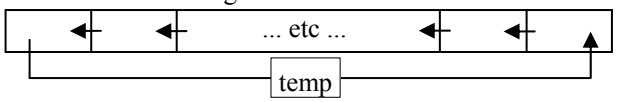
Participants were unanimous that this question was too open, ambiguous, and poorly specified. Some felt that it might be suitable for a practical programming test, but none thought it suitable for a written exam.

Q28, on the other hand, was considered to be difficult but usable. None of the participants expressed concern about the assumption that left represents the lower indexes of the array and right represents the upper indexes – an assumption that is supported by the diagram. The general response was approval (especially when the explicit '5' was removed from the first sentence). The participants liked this question, at the same time acknowledging that this was one of the most difficult questions in the set. That is, they tended to agree with the

Q28. The purpose of the code below is to take an array of numbers (values) containing 5 integers and move all elements of the array one place to the left, with the leftmost element moving to the rightmost position.

```
temp = values[0];
for (int i=0; i<values.Length-1; i++)
    values[i] = values[i+1];
values[values.Length – 1] = temp;
```

For example, if *values* initially has the value [1, 2, 3, 4, 5], then after the code has executed, it would contain [2, 3, 4, 5, 1]. If we were to show the effect of moving all the elements of an array in this way in a diagram, it would look something like this:



Write code that does the opposite of the original block of code above. That is, write code to move all elements of the array values one place to the *right*, with the *rightmost* element being moved to the *leftmost* position.

unspoken notion that an exam should include a mix of easy, medium, and hard questions, and that this question could be one of the last group. Nevertheless, in a subsequent project to use the selected questions in a number of final exams, one instructor decided that the improved version of this question was too difficult and could not be used. It is not clear whether the question was considered too hard even to be one of the exam's more difficult questions, or whether that instructor chooses not to include any difficult questions in exams.

3.7 Form of the question

Most of the exams studied by Sheard et al (2013) included a mix of multiple-choice, short-answer, and code-writing questions, and our question set included examples of all three types.

One issue that does not yet seem to have been addressed in the literature is whether different forms of the same question are equivalent. Our study explicitly addressed this question by including three different forms of the same question, Q29.

Most participants liked the code-writing form of the question, Q29a, with the qualification that some courses might prefer the word 'position' to 'index'.

Q29b, filling in the blanks, was regarded much less favourably. One participant saw it as a trick question that encouraged the students to copy code directly from the first listing to the second, especially as it omits the description of the difference, that is, that the first piece remembers the element while the second should remember the index. Another participant felt that this version was an improvement, removing the potentially confusing wording. A third simply said that students would be horribly confused by this question, while a fourth thought that it might be better as a Parsons problem (Parsons & Haden 2006) – presumably the variant in which multiple options are available for each line of code, as otherwise it could be solved trivially by comparison with the preceding listing.

Q29a. The following piece of code sets *answer* to the smallest element of the integer array *num*.

```
int best = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < best) best = num[i];
}
answer = best;
```

This code works by remembering, in *best*, the value of the smallest element met so far as it works through the array. Write a piece of code that achieves exactly the same outcome, setting *answer* to the smallest element of *num*, but by remembering *the index of* the smallest element met so far.

Q29b. The following piece of code sets *answer* to the smallest element of the integer array *num*.

```
int smallest = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < smallest)
  {
    smallest = num[i];
  }
}
answer = smallest;
```

Complete the code in the boxes below so that it also sets *answer* to the smallest element of *num*. Note that the sixth line is different in the two listings.

```
int where = [          ];
for (int i=0; i < num.Length; i++)
{
   if num[i] < [          ])
   {
     where = i;   // Note difference
   }
}
answer = [          ];
```

Q29c. The following piece of code sets *answer* to the smallest element of the integer array *num*.

```
int best = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < best) best = num[i];
}
answer = best;
```

Which of the following pieces of code does exactly the same thing, that is, sets *answer* to the smallest element of *num*?

(a)
```
int best = 0;
for (int i=1; i < num.Length; i++)
{
  if (num[i] < num[best]) best = i;
}
answer = num[best];
```

(b)
```
int best = 0;
for (int i=1; i < num.Length; i++)
{
  if (num[i] < num[best]) best = num[i];
}
answer = num[best];
```

(c)
```
int best = 0;
for (int i=1; i < num.Length; i++)
{
  if (num[i] < num[best]) best = i;
}
answer = best;
```

(d)
```
int best = num[0];
for (int i=1; i < num.Length; i++)
{
  if (num[i] < num[best]) best = i;
}
answer = num[best];
```

The multiple-choice version, Q29c, was seen by one participant as the best of the options. On the other hand, three believed that it would be too easy to find the answer by strategic guessing or reverse engineering as opposed to reading and understanding the four different pieces of code. It remains an open question whether the strategic guessing or reverse engineering would require students to reason in a similar way as they would if reading and understanding the code pieces, in which case there might not be a problem.

In addition to asking whether participants would use each version of this question in their exams, we asked whether they thought that the three versions were the same, and why.

Nobody thought that they were the same. One participant thought they were equivalent, "essentially but not exactly" the same, and some noted that they were testing the same thing in different ways. Others, however, felt the versions to be quite different as they test different skills: code writing, scaffolded code writing, and code tracing. Most participants thought the multiple-choice version to be the easiest, but one thought that the pure-code writing version was easiest, and two favoured the scaffolded code-writing version.

3.8 Multiple-choice questions

Multiple-choice questions have been the subject of much discussion in the literature, essentially addressing the question of whether they are a legitimate form of assessment. There are guides to writing good MCQs (Hansen 1997, Isaacs 1994), a number of papers proposing how MCQs can be validly used in computing assessment (Lister 2005, Roberts 2006, Woodford & Bancroft 2005), but at least one survey showing that many instructors remain highly suspicious of this question form (Shuhidan et al 2010).

Some participants in our study echoed this suspicion. Of the 33 questions in the study, 11 were presented in the multiple-choice form, and all but three of those drew suggestions that the answers would be too easy to guess, requirements to add further distractors, or both. Some participants who normally use MCQs in their exams expressed no such concerns, but this form of question is clearly still worrying to many instructors.

3.9 Code-explaining questions

A number of the questions in this study ask students to explain the purpose or outcome of a given piece of code.

Q14. Consider the following block of code, where variables *a*, *b*, and *c* each store integer values:

```
if (a > b) {
   if (b > c) {
      Console.WriteLine(c);
   } else {
      Console.WriteLine(b);
   }
} else if (a > c) {
   Console.WriteLine(c);
} else {
   Console.WriteLine(a);
}
```

In one sentence, describe the purpose of the above code (i.e. the if/else if/else block). Do NOT give a line-by-line description of what the code does. Instead, tell us the purpose of the code.

Q19 in section 3.1 and the hypothetical Q∞ in section 3.4 are examples; Q14 is another.

Code-explaining questions were brought into wide use by the BRACElet project (Whalley et al 2006), to test the notion that perhaps students should be able to read code before they can be expected to write code. That project consistently found that introductory programming students had great difficulty deducing the purpose of small pieces of code (Sheard et al 2008, Teague & Lister 2014), even if the questions were presented in multiple-choice form (Simon & Snowdon 2011).

The greatest concern expressed by participants about these questions is their use of non-meaningful variable names. However, as discussed in section 3.4, it would be difficult to provide meaningful variable names without giving away the purpose of the code. Therefore it would seem that neutral variable names might be an unavoidable cost associated with using questions of this type.

With code-explaining questions, as with other questions, it is important to avoid obfuscation. The point can be illustrated with Q14. A knowledgeable programmer might respond that the code prints the smallest value of the variables *a*, *b*, and *c*. Others, however, might wonder how to describe what will happen if two or three of the variables are equal. Would that notion of 'smallest' then strictly apply, and if not, how should they describe which of the equal variables would have its value printed? It is unlikely that these questions were considered by the question's author, yet they have the potential to seriously confuse some students.

Is there, then, any point in setting code-explaining questions? Many appear to think so, and the participants in this study certainly expressed general approval of some of the code-explaining questions provided.

One point that was clearly made by the BRACElet project is that students are less likely to do well on code-explaining questions if they are not familiar with this question type. A final examination is seldom the best place to introduce students to a type of question they have not seen before. Instructors deciding to introduce code-explaining questions to their exams should certainly give students ample prior practice with this type of question.

4 Results: ten questions for broad use

On a scale from 1 (would definitely not use) to 5 (would definitely use), the 33 questions were accorded average ratings ranging from 2.9 (Q2, discussed in section 3.6) to 4.9 (Q5, discussed in section 3.5). Fourteen of the questions, nearly half of them, rated at 4 or above, and only five rated below 3.5.

When participants ranked a question less than 5, their comments sometimes made it clear that they would be happy to use the question with suitable amendments.

We selected ten questions, working from the highest-ranked, so as to produce a mix of question styles and topics. The lowest-ranked question that we selected had an average of 3.6, but was substantially altered (for example, changing it from multiple-choice to short-answer type) to address some of the concerns expressed; the question would therefore have rated more highly if it had been presented in this altered form. All of the other questions chosen had average ratings of 3.9 or higher.

All ten questions are presented in the appendix.

5 Results: how (not) to write an introductory programming exam

The ratings given to the various questions in our study, and the discussion on whether the participants would use each question, lead to a set of guidelines that can be used when writing an exam. The guidelines can be used as a set of positive recommendations, or used in their converse forms as a set of practices to be avoided. Some of these guidelines are already well known, but we believe that there is value in presenting them here as a full set.

Keep questions as simple as possible. Unless you are deliberately making a question complex to test your students' skills in gathering requirements and solving problems, simplify question preambles as much as you possibly can. Then check them to see if you can simplify them still further. Finally, have some colleagues check them, to be sure that they interpret them the same way you do. Include questions in a range of difficulty levels, but be sure that the difficulty of a question is germane, deriving from the inherent difficulty of the task to be performed, not from difficulty in understanding what that task might be.

Consider not contextualising questions. If it is your preference to provide a little real-world (or pretend-world) context for your exam questions, consider whether that context might in fact tend to confuse or mislead students. If it might, consider removing the context so that students will focus on the question you are actually asking.

Use diagrams and examples to help students understand the question. This comes back to the question of what is germane. If it is your goal to see whether students can answer the question, do everything you can, within reason, to ensure that the students understand what the question is. If a diagram or example seems more likely to help students than to further confuse them, provide one. A diagram is far less likely to confuse students if they have seen a number of similar diagrams during the course.

Ensure that students are familiar with the types of question used. It is good to consider adding new question types to an exam, but it might be unfair on the students if the exam is the first place that they see questions of this type. Try to ensure that they have prior exposure to each type of question used in the exam.

When providing code as part of a question, write it as you have taught the students to write. If you have spent a semester trying to teach the students to use good programming style, do not present them with code written in poor style. The exception to this is that neutral variable names should be used if meaningful variable names would give away the answer in a code-explaining or code-tracing question.

Avoid variable names that are easily confused with one another or with other symbols. Consider the ease of confusing *p* and *q*, *b* and *d*, *i* and 1, *l* and 1, *O* and 0; wherever possible, avoid using these single-letter variable names.

Eschew obfuscation. Do not deliberately complicate code. Your exam should determine who can read and understand well-written code – not who can unscramble code that has been written poorly. That skill might be better left for a course on code maintenance.

Include questions of a range of difficulties. Have some easy questions, some moderate questions, and some difficult questions. Easy questions give almost all students a chance to show that they know something about what was taught. Difficult questions, preferably not weighted too heavily, help to distinguish the best students from the rest of the class.

Consider including some multiple-choice questions. It really is possible to write MCQs that test skills other than memory recall, and that distinguish well between the poor students and the good students. They are definitely easier to mark than written-answer questions. And while bright students might be able to deduce the answers by some form of elimination, these are the students who don't need to do so, because they can answer the questions in the way that was intended. Despite the concerns of some of our participants, many students do select wrong answers to MCQs.

Consider including some code-reading questions. Do not assume that your students can read and understand code simply because in a code-writing question they can cobble together an approximation to the answer you were expecting. Be prepared to explicitly test their code comprehension skills.

Include questions of different forms. Be aware of the many different types of question that can be used in an exam, and consider which question type is best suited to each question you intend to ask. Be aware that the same question in different forms is likely to be testing different skills, and choose the form that tests the skills you wish to assess.

6 Conclusions

We set out to answer three questions. Our results show that all three questions can be answered in the affirmative.

Can we identify some principles of good question design that others can apply in writing their own questions? Can we identify some aspects of poor question design that others can try to avoid when writing their own questions? We can and we have. The guidelines in section 5 should be useful to anyone writing an exam, not just in introductory programming but in programming at any level, though of course matters such as question difficulty will need to be adjusted for higher-level courses. Some of the guidelines extend beyond programming, and apply to exam writing in general.

Can we identify examination questions that a group of instructors would all be willing to use in their introductory programming exams? We can and we have. The questions provided in the appendix have been selected on the basis of evaluation by nine academics involved with the assessment of introductory programming courses.

We invite others to include the questions in their own exams, and to either join us in publishing the results, or simply to compare their own students' performance with the benchmark results that we expect to publish. The versions in the appendix are all written in Java, but the project leaders can supply versions of the same questions in C, C#, Visual Basic, Python, and TouchDevelop, and are willing to work on versions for other suitable languages if required. However, we hope it is clear that the questions are not suited to all programming languages, and in particular that they are unlikely to be usable in courses that teach using a functional language and approach.

7 References

Ahadi, A., and Lister, R. (2013). Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant? Ninth International Computing Education Research workshop (ICER 2013), 123-128.

Guzdial, M.J. and Ericson, B. (2013). Introduction to Programming and Computing in Python: a Multimedia Approach, 3rd edition, Pearson Education Inc.

Hansen, J.D. and Dexter, L. (1997). Quality multiple-choice test questions: item-writing guidelines and an analysis of auditing testbanks. Journal of Education for Business 73(2):94-97.

Isaacs, G. (1994). Multiple choice testing. HERDSA Green Guide No 16. Higher Education Research and Development Society of Australasia Inc, Campbelltown, Australia.

Lister, R. (2005). One small step toward a culture of peer review and multi-institutional sharing of educational resources: a multiple choice exam for first semester programming students. Seventh Australasian Computing Education Conference (ACE2005), 155-164.

Lister, R., Corney, M., Curran, J., D'Souza, D., Fidge, C., Gluga, R., Hamilton, M., Harland, J., Hogan, J., Kay, J., Murphy, T., Roggenkamp, M., Sheard, J., Simon, and Teague, D. (2012). Toward a shared understanding of competency in programming: An invitation to the BABELnot project. 14th Australasian Computing Education Conference (ACE 2012), 53-60.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A multi-national, multi-institutional study assessment of programming skills of first-year CS students. SIGCSE Bulletin, 33(4):125-140.

Parsons, D. and Haden, P. (2006). Parson's programmimg puzzles: a fun and effective learning tool for first programming courses. Eighth Australasian Computing Education Conference (ACE 2006), 157-163

Petersen, A., Craig, M., and Zingaro, D. (2011). Reviewing CS1 exam question content. 42nd ACM Technical Symposium on Computer Science Education (SIGCSE 2011), Dallas, Texas, USA.

Roberts, Tim (2006). The use of multiple choice tests for formative and summative assessment. Eighth Australasian Computing Education Conference (ACE2006), 175-180.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., and Whalleyt, J. (2008). Going SOLO to assess novice programmers. 13th Conference on Innovation and Technology on Computer Science Education (ITiCSE 2008), 209-213.

Sheard, J., Simon, Carbone, A., Chinn, D., Clear, T., Corney, M., D'Souza, D., Fenwick, J., Harland, J., Laakso, M.-J., and Teague, D. (2013): How difficult are exams? A framework for assessing the complexity of introductory programming exams. 15th Australasian Computing Education Conference (ACE 2013), 145-154.

Sheard, J., Simon, Dermoudy, J., D'Souza, D., Hu, M., and Parson, D. (2014). Benchmarking a set of exam questions for introductory programming. 16th Australasian Computing Education Conference (ACE 2014), 113-121.

Shuhidan, S., Hamilton, M., and D'Souza, D. (2010). Instructor perspectives of multiple-choice questions in summative assessment for novice programmers. Computer Science Education 20(3):229-259.

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J., and Warburton, G. (2012). Introductory programming: examining the exams. 14th Australasian Computing Education Conference (ACE 2012), 61-70.

Simon and Snowdon, S. (2011). Explaining program code: giving students the answer helps – but only just. Seventh International Computing Education Research Workshop (ICER 2011), 93-99.

Simon, B., Clancy, M., McCartney, R., Morrison, B., Richards, B., and Sanders, K. (2010). Making sense of data structures exams. Sixth International Computing Education Research workshop (ICER 2010), 97-105.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. Communications of the ACM, 29(9), 850-858.

Teague, D. and Lister, R. (2014). Blinded by their Plight: Tracing and the Preoperational Programmer. 25th Psychology of Programming Interest Group Annual Conference (PPIG 2014).

Traynor, D., Bergin, S., and Gibson, J.P. (2006). Automated assessment in CS1. Eighth Australasian Computing Education Conference (ACE 2006), 223-228.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P.K.A., and Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. Eighth Australasian Computing Education Conference (ACE 2006), 243-252.

Woodford, K. and Bancroft, P (2005). Multiple choice questions not considered harmful. Seventh Australasian Computing Education Conference (ACE2005), 109-115.

Appendix: the ten selected questions (renumbered for subsequent use)

Q1. If a dependent child is a person under 18 years of age who does not earn $10,000 or more a year, which expression would define a dependent child?

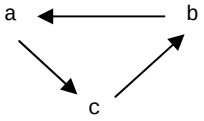
(a) `age < 18 && salary < 10000`
(b) `age < 18 || salary < 10000`
(c) `age <= 18 && salary <= 10000`
(d) `age <= 18 || salary <= 10000`

Q2. What are the values of *girls*, *boys*, and *children* after the following code has been executed?

```
int girls = 0;
int boys = 0;
int children = 0;
children = girls + boys;
girls = 15;
boys = 12;
```

(a) 0, 0, 0
(b) 0, 0, 27
(c) 15, 12, 0
(d) 15, 12, 27

Q3. There are three integer variables, *a*, *b* and *c*, which have been initialised. Write code to shift the values in these variables around so that *a* is given *b*'s original value, *b* is given *c*'s original value, and *c* is given *a*'s original value. The following diagram illustrates the direction of the shifts:



Q4. What will be the value of the variable *z* after the following code is executed?

```
int x = 1; int y = 2; int z = 3;
if (x < y) {
    if (y > 4) {
        z = 5;
    } else {
        z = 6;
    }
}
```

Q5. Consider the following block of code, where variables *a*, *b*, *c*, and *answer* each store integer values:

```
if (a > b) {
    if (b > c) {
        answer = c;
    } else {
        answer = b;
    }
} else if (a > c) {
    answer = c;
} else {
    answer = a;
}
```

Which of the following sets of values for *a*, *b*, and *c* will cause *answer* to be assigned the value in variable *b*?

(a) a = 1, b = 2, c = 3
(b) a = 1, b = 3, c = 2
(c) a = 2, b = 1, c = 3
(d) a = 3, b = 2, c = 1

Q6. What will be the value of *result* after the following code statements are executed?

```
int[] nums1 = { 1, -5, 2, 0, 4, 2, -3 };
int[] nums2 = { 1, -5, 2, 4, 4, 2, 7 };
int result = 0;
int j = 0;
while (j < nums1.length)
{
    if (nums1[j] != nums2[j])
    {
        result = result + 1;
    }
    j = j + 1;
}
```

Q7. What is the outcome or likely purpose of the following piece of code?

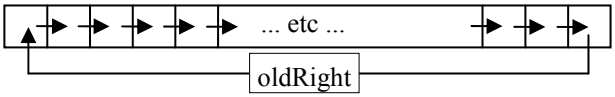
```
int result = 0;
for (int j = 0; j < number.length; j++)
{
    if (number[j] < 0)
    {
        result = result + 1;
    }
}
```

(a) to find the smallest number in the array
(b) to count the negative numbers in the array
(c) to sum the negative numbers in the array
(d) to add 1 to each of the negative numbers in the array
(e) to find the index of the first negative number in the array

Q8. What is the outcome or likely purpose of the following piece of code? Express your answer as a short phrase, like the phrases provided as possible answers in question 7.

```
int result = 0;
for (int count = 1; count <= num; count++)
{
    result = result + count;
}
```

Q9. We can represent an array of integers as a sequence of elements arranged from left to right, with the first element at the left and the last element at the right. Using this representation, a programmer wishes to move all elements of an array one place to the right, with the rightmost element being 'wrapped around' to the leftmost position, as shown in this diagram.



Here is the code that performs that shift for an array referred to by the name *values*:

```
int oldRight = values[values.length - 1];
for (int j = values.length - 1; j > 0; j--)
    values[j] = values[j - 1];
values[0] = oldRight;
```

For example, if *values* initially contains the integers [1, 2, 3, 4, 5], once the code has executed it would contain [5, 1, 2, 3, 4]. Write code that will undo the effect of the above code. That is, write code that will move all the elements of the array one place to the left, with the leftmost element being wrapped around to the rightmost position.

Q10. Write a method that will be given an array of integers and will calculate and return (as a double) the mean (average) of all the integers in the array.