Code or (not Code) – Separating Formal and Natural Language in CS Education

Quintin Cutts
School of Computing Science
University of Glasgow
Glasgow, Scotland
+44 141 330 5619
Quintin.Cutts@glasgow.ac.uk

Richard Connor
Computer & Information Sciences
University of Strathclyde
Glasgow, Scotland
+44 141 548 3424
Richard.Connor@strath.ac.uk

Peter Donaldson
Dept. of Computing Science
Crieff High School
Crieff, Scotland
+44 1764 657600
PWDonaldson@pkc.gov.uk

Greg Michaelson
Dept. of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh, Scotland
+44 131 451 3422
G.Michaelson@hw.ac.uk

ABSTRACT
This paper argues that the "institutionalised understanding" of pseudo-code as a blend of formal and natural languages makes it an unsuitable choice for national assessment where the intention is to test program comprehension skills. It permits question-setters to inadvertently introduce a level of ambiguity and consequent confusion. This is not in keeping with either good assessment practice or an argument developed in the paper that CS education should be clearly fostering the skills needed for understanding *formal*, as distinct from *natural*, languages. The argument is backed up by an analysis of 49 questions drawn from the national school CS examinations of a single country, spanning a period of six years and two phases – the first in which no formal pseudo-code was defined, the second in which a formal reference language, referred to as a "formally-defined pseudo-code", was provided for teachers and exam setters. The analysis demonstrates that in both phases, incorrect, confusing or ambiguous code was presented in questions. The paper concludes by recommending that the term *reference language* should be used in place of *pseudo-code*, and an appropriate formally-defined language specified, in national exam settings where a common language of assessment is required,. This change of terms emphasises the characteristics required of a language to be used for assessment of program comprehension. The reference language used in the study is outlined. It was designed by the authors for human readability and also to make absolutely explicit the demarcation between formal and informal language, in such a way that automated checking can be carried out on programs written in the language. Formal specifications and a checker for the language are available.

WiPSCE '14, November 05 - 07 2014, Berlin, Germany
Copyright 2014 ACM 978-1-4503-3250-7/14/11…$15.00
http://dx.doi.org/10.1145/2670757.2670780

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

General Terms
Human Factors, Languages.

Keywords
assessment, comprehension, language, program, pseudo-code, reference

1. INTRODUCTION
The importance of program comprehension (PC) has been raised in many studies as a key developmental step towards learning how to program. Such work has typically concentrated on two key aspects:

- Connecting localised elements of the code through to the concepts and mechanisms that underpin the programming language, e.g. [16].
- Appreciating higher-level idiomatic use of the low-level language constructs, maybe spanning multiple lines, and variously called *plans, chunks, patterns,* e.g. [6,7,13,18].

This paper argues that a third aspect should also be considered:

- Recognising that the processes used to understand texts written in formal and natural languages are significantly different.

This third aspect is proposed because programming languages are in the category of formal languages whose syntax contains very limited redundancy and whose sentences each have *one single meaning* in accordance with the programming language definition. This is in stark contrast to natural languages with their high levels of redundancy and sentences that can often be ambiguous. The comprehension skills for the two kinds of language are therefore necessarily different, as will be explored in the paper.

This difference in formal and natural languages and the consequent differences in comprehension styles should be made clear in our educational programmes, along with specific guidance on how to approach the understanding of formal languages. Although noted as the third aspect above, it is really the first one,

as it underpins the ability to address the others. One might think that a mathematical background would be a good foundation for the right kind of reading skills, given mathematics' use of formal language and exposure to it from an early age. However, this comment from an undergraduate university student, studying on a computational thinking skills course and reported in [5], indicates that this is not always the case:

I feel that learning the language of computing definitely helps you understand dense reading a lot more efficiently. I personally have noticed that my in-depth understanding of Computer Science wording has helped me understand my mathematical theorems and proofs more regularly than before

The course that elicited this student response pays explicit attention to code comprehension, requiring students to practice the skill repeatedly in every class session.

The emerging emphasis on PC skills in CS education suggests that appropriate assessment of these skills is required. In particular, the expansion of CS education into the school sector (K-12) requires the discipline to consider how national examinations will manage this aspect of assessment.

One emerging route for such assessment, in contexts where examining authorities choose not to mandate a single programming language for instruction and assessment, is to use a *formalised pseudo-code*, e.g. [1,14]. Pseudo-code is typically considered to be a blend of formal and natural languages, used for human understanding of algorithms rather than machine understanding, and so it is easy to understand this choice. In different contexts, pseudo-code can be more or less formal. For example, in text-book descriptions of algorithms, the pseudo-code is mostly formal, with the particular dialect being defined at the start and then used consistently throughout the book; when used in its other major role, as a notation for developing programs, it is often much less formal, varying widely between different programmers.

This paper argues that the "institutionalised understanding" of pseudo-code as a blend of formal and natural languages may make it an unsuitable choice for national assessment where the intention is to test program comprehension skills. It permits question-setters to inadvertently introduce a level of ambiguity and consequent confusion. This is not in keeping with either good assessment practice or the argument developed here that CS education should be clearly fostering the skills needed for understanding *formal*, as distinct from *natural*, languages.

The argument is backed up by an analysis of 49 questions drawn from the national school CS examinations of a single country, spanning a period of six years and two phases – the first in which no formal pseudo-code was defined, the second in which a formally-defined reference language, referred to as a "formal pseudo-code" was provided for teachers and exam setters. The analysis demonstrates that in *both* phases, incorrect, confusing or ambiguous code was included in questions.

Note that the issue is not whether a student will be able to understand the code written by the question setters. It is that the setters did not understand the requirements placed upon them when using a formally-specified reference language, continuing to use unspecified natural language constructs, completely at odds with a test of PC skills.

The paper concludes by suggesting that the term *reference language* should be used in place of *pseudo-code* in national exam settings where a common language of assessment is required, along with necessary training. The reference language used in the study was developed by the authors, addressing both human readability and making absolutely explicit the demarcation between formal and informal language, in such a way that automated checking can be carried out on programs written in the language. Formal specifications and a checker for the language are available.

The contribution of the paper is therefore as follows:

- A theoretical argument that CS educators should focus on the fundamental differences between formal and natural languages at an early stage in the fostering of program comprehension skills (in Sections 3 and 4).
- An empirical study of the use of pseudo-code in a national CS exam system, suggesting that it is an unsuitable format for assessing program comprehension (Sections 5 and 6).
- The outline features of a reference language for use in assessing program comprehension, which has a full implementation, to foster discussion in this area (Section 7).

2. RELATED WORK AND BACKGROUND

Shulte et al.'s survey article [13] on PC considers a number of PC models, setting an analysis of each against Schulte's Block Model [12], itself an educational model of PC. The Block Model considers understanding at four levels of detail within a program, from individual language elements, through blocks of code consisting of adjacent statements and then multiple related blocks, up to the whole program. For each of these levels of detail, understanding is further categorized into two *structural* elements, according to its appearance in a program text and how it operates during execution, and a *functional* element - how it contributes to the overall goals of the program in respect of solving a particular problem.

The survey considers how prior research on PC has influenced CS education. Borstler et al. [3] are cited as follows: "There is a large body of knowledge on program comprehension… but this is rarely applied in an educational setting." The survey suggests that this may be because the "focus of much of the prior PC research has been on the work of professional programmers rather than students", and also because of the strong focus on construction of programs, not comprehension, in most introductory programming classes.

Furthermore, in drawing out goals for education, the survey notes "most PC models emphasize the importance of understanding based on the program code (i.e. reading). Perhaps it should be explicitly emphasized in education as well", by including reading and comprehension strategies in courses. Beyond the sole example of reading in the sequence of the execution (and not linearly through the program text), drawn from Fix et al. [6], little advice is given of what such reading strategies might look like. In highlighting areas for further research, they note "specifically, it seems a neglected topic to teach suitable program reading strategies."

Sorva [16] highlights the importance of developing an understanding of the notional machine that is embodied in the programming language definition. Indeed, the structural element of the Block Model, involving the operation of programming language elements, directly relates to this idea of a notional machine.

Soloway and Ehrlich's study of expert and novice ability to understand code [18] draws on the idea of a text "schema", from text processing research in AI and psychology. They write that schemas are "generic knowledge structures that guide the comprehender's interpretations, inferences, expectations, and attention when passages are comprehended." The study determined that the use of *good rules of discourse*, that is, programming idioms or schemas widely accepted across the discipline, underpinned experts' increased ability to understand programs compared to novices. When programs were written using unusual programming idioms, experts' and novices' comprehension abilities were the same. The use of regular idioms was a core feature of Soloway's approach to learning to program [19]. Although Soloway does not consider teaching of code comprehension directly, it seems likely that programs used for code comprehension activities should employ the good rules of discourse he identifies.

Fix et al. [6] highlight five key attributes of program comprehension, conducting studies comparing experts' and novices' performance on these. The most relevant here is "grounding in the program text", referring to a programmer's ability to map from aspects of their mental representation of the program to where those aspects exist in the program text. Experts were significantly better than novices on two of the three measures used in the "grounding in the program text" attribute. The authors consider the importance of particular reading strategies for novices, noting that "a different reading or study strategy may obscure some information selected by experts and at the same time may highlight the information less useful to support programming tasks."

Lopez et al [8] argue for the need to attend to PC as an essential developmental step towards being able to write programs. Lister's later work [7] on relating Neo-Piagetian theory to the development of programming skills explores the developmental steps for students who have mastered code tracing and therefore have some level of facility with the Structural level of the Block Model.

Soloway et al. [17] identify the challenge of novices' transferring the meaning of words in natural languages across to the same words used as tokens in formal languages, when in fact the meanings are different. While this is not directly addressed in this paper, being explicit about natural and formal languages in educational programmes should help.

This short survey identifies the key importance of understanding the operational model defined by the program itself and of the manner in which the components of that model link across to the problem domain; it highlights the need for using regular idioms in programs to aid comprehension; it suggests that the development of reading and comprehension strategies are currently poorly served in education programmes; and it notes that how students interpret formal languages influences their programming ability.

3. FORMAL VS. NATURAL LANGUAGE IN PROGRAMMING

This section and the next argue that the fundamental difference between formal and natural languages should be made clear in CS education at an early stage. First, these differences are outlined.

Formal languages are fully defined. That is, all valid sentences in a formal language can be derived from a set of axioms, or atoms, and the application of a set of rules over these axioms. Any valid sentence has *precisely one* meaning according to this definition of axioms and rules. The meaning referred to here has no connection to any external context – there is only the language definition (and, following Sorva, the notional machine that it defines) and the particular use of symbols within any given program.

This understanding of the meaning of a program relates directly to the *structural* aspect of the Block Model, involving the appearance of program elements in programs and their operation or execution at the level of Sorva's notional machine concept.

By comparison, the use of natural languages involves making assumptions that the meaning of language entities is understood on the basis of a good awareness of external context. That is, the meaning of a sentence is clear not purely on the basis of the symbols contained within the sentence, but only when the surrounding context is also considered. For example, the natural language instruction *Go over there* can only be fully understood if there is some way of resolving the ambiguity of what *there* means. If this is a spoken instruction between two people, then the speaker's nod or a pointed finger is the necessary external information required by the receiver to properly carry out the task.

This paper is primarily concerned with the mental processes involved in understanding language. On the basis of the differences between formal and natural languages outlined above, the exercise of understanding sentences in each kind of language requires two quite different mental processes:

- For a formal language, a single and complete meaning is contained entirely within the text, and the understanding process consists of determining that meaning from a close analysis of the text alone.
- For a natural language, an analysis of the text is only part of the task, as this may produce multiple possible meanings. A particular meaning can only be derived by making use of available contextual information for disambiguation.

Furthermore, natural languages contain significant redundancy, enabling meaning to be determined even when particular elements of a sentence are missed, for example when speaking or reading fast or over a noisy channel. Reading involves immediately throwing away the redundant words and concentrating on those words that deliver the sentence's meaning. Listening to someone speaking, in a lecture for example, can be frustrating because so much of what they say is actually redundant.

By comparison, formal languages tend to contain very little redundancy, and therefore almost every symbol in a sentence contributes directly to the meaning of that sentence. Fast reading can easily lead to important information being missed out, resulting in an incorrect understanding being derived.

The Block Model of program comprehension defines both a *structural* and *functional* understanding of a program. As already noted, the construction and interpretation of a program code from the formal language standpoint relates to the Block Model's structural understanding. However, programs are designed to solve problems defined in a domain external to that of the formal language, and the understanding of programs against this external domain is the Block Model's functional understanding. Siebel's experience of running a code-reading group underlines this [15].

The challenge of this duality is that natural language is overlaid onto the formal programming language in order to facilitate this functional understanding. In particular, identifiers and program commentary making use of natural language provide a link from the self-contained programming language domain out to the problem domain. This overlaying of natural language onto the

formal language may lead novices to adopt an inappropriate strategy for understanding programs.

4. EMPHASISING THE USE OF FORMAL LANGUAGES IN CS EDUCATION

The general lack of focus on teaching program comprehension reported in the literature ("a neglected topic"[13]), should be a concern for CS educators, yet it is perhaps understandable given that nearly all the findings in this area come from observations *of* experienced programmers *by* experienced programmers. Maybe we are simply missing some key underpinnings to program comprehension because they are so automatic to us.

By analogy, the evaluation of an introduction to computational thinking course reported in [5] notes that one of the major developmental steps reported by complete novices in the class was to understand that programs are *deterministic* and hence that computers do exactly what you tell them to do. This is such a fundamental understanding for a computational thinker that as educators, we may never have considered we needed to teach such a concept explicitly, assuming that all would already know this.

Similarly, consider once again that programs written in a formally defined programming language have a single meaning derived from the axioms and rules of the language definition and the particular symbols and constructs used in the program, independent of any external context. This understanding may be a universally held and therefore unexamined tenet among experienced programmers.

Indeed, the lack of these two key understandings, which are coincidentally closely related, can be seen as contributors to the "superbug" identified by Pea through analysis of novices' bugs, that is, the assumption that the programming language system or machine has some kind of intelligence or external wisdom that will interpret the words of a program in the way the programmer intended. [10]

In the light of the analysis here of how texts in different language types are understood, we can postulate that a student with the superbug misconception sees a program as a text in natural language, the language style with which they are most familiar, and ascribes to the machine the same facility that they personally possess to understand texts in natural language. That is, the ability to call on contextual information, in this case, the intention in the student's head.

Another reason why we as educators have perhaps not attended to this issue before is that the students we see are primarily self-selected and a majority at least have these core understandings in place. The nature of formal languages is often not introduced until later courses, e.g. a theory of computation course. However, we are moving towards an era where computational thinking skills are being recommended for all, and we need now to ensure that all-comers can succeed, irrespective of their particular background that may or may not have given them the appropriate core understanding. We need some theory of languages early on.

Given that the superbug represents the lack of the kind of conceptual understanding inherent in a *threshold concept* [9] or crucial to maintaining *learning edge momentum* [11], it is essential that it is addressed as early as possible in a programming course, and that learning designs maintain an emphasis on the Block Model's structural aspect of program comprehension as well as the functional aspect.

5. ASSESSING PC SKILLS

If the teaching of PC skills is increasing in importance, then the assessment of those skills should be considered also. This paper has so far argued for the value in distinguishing formal and natural languages at the earliest stage in an effort to ensure that a misunderstanding of the nature of program text does not hamper progress. Therefore assessment of PC skills, in alignment with this emphasis on distinguishing formal and natural language, should be included from the earliest stages too.

PC assessment involves presenting code to students and asking them to answer questions about the code. The key issue for the remainder of this paper is which, or what kind of, language should be used to present the code. There are three recurring approaches, the first of which is as follows:

- In a localised context, where the teacher is also the assessor, the programming language of instruction can also be the language of assessment. This is the general context for all university CS education. This approach can also be used for national qualifications if the awarding body trusts local teachers to both set and mark assessments internally and are content for any programming language to be used. This is the approach taken in the recently introduced New Zealand school curriculum [2].

The second two approaches to national assessments stem from the exam board's setting a single exam to be taken across the nation, as follows:

- The exam board specifies a single programming language for assessment, and so all schools using the board's examinations are likely to adopt that language for instruction too. The US Advanced Placement Computer Science course adopts this approach, using Java as the specified language [4].
- The exam board specifies a *pseudo-code* that will be used in lieu of any one programming language, thus freeing up schools to use whatever language of instruction they choose. This is an attractive proposition when it is known that the preferences for and confidence with particular programming languages vary widely across the teacher population. This approach is taken by exam boards in England and Scotland [1,14].

With the current trend in the western world towards fostering a level of computer science education for all in the school sector, understanding the consequences of these different approaches is important. In the first two approaches, programming languages in the class of formal languages are used for assessment of PC skills, and so represent a valid choice for assessment in the context of this paper.

The third approach makes use of a semi-formal language, in that the pseudo-code has a more or less clear definition. Furthermore, this semi-formal language is associated by name (pseudo-code) with a language type that is typically viewed as informal and understood by context, and that makes use of both formal and natural language elements.

Of interest in this paper, then, is the following question: does the code used in PC exam questions prepared using more or less well-defined pseudo-code represent a valid context for assessing program comprehension skills, given the paper's emphasis on clearly demarcating formal and natural language? *This is an evaluation of those creating examination questions*: are question setters presenting code in a manner that ensures formal language

comprehension skills are being assessed? Breaking this down into two questions:

1. Are the core computational constructs presented consistently?
2. If natural language descriptions are used, are they clearly demarcated from the use of the formal programming language constructs?

If these questions can be answered positively, then question setters are using the language in a manner appropriate to supporting a valid assessment. If the questions cannot be answered positively, then question setters are basing their use of language on traditional natural language characteristics rather than from the context of a formal language.

Crucially, note that it is entirely immaterial whether a student can or cannot understand the code presented. This paper is about teaching and examining *formal* language comprehension skills. If natural language elements are mixed freely with the formally-defined language, then, even if the student answers correctly, a different set of skills have been assessed.

The next section presents an analysis of 49 exam questions drawn from the CS qualifications of a national exam board spanning a six-year period, in order to shed light on how question-setters use pseudo-code in exam questions.

6. EVALUATING THE USE OF PSEUDO-CODE IN ASSESSMENT

6.1 Study Context

Scotland has been running nationwide high school courses in computer science for around 30 years. The examinations authority experienced early difficulties with mandating a single language for all schools to use, because in the 1980s a wide range of machines was used in schools with the consequent difficulty of ensuring that all schools had access to the single prescribed language. Hence the examinations authority have chosen, then and now, not to mandate a single language for use in written examinations marked externally to the school where they were taken. Should a candidate be asked to write a program in a written examination, they are permitted to use any language with which they are familiar, and individual schools can use the programming language of their choice for both instruction and internally-assessed coursework components.

As with most programming courses, those in Scotland have largely focussed on the creation of programs, rather than whether a student can understand and analyse programming language code. In national exams, pseudo-code has appeared over this period, both to describe algorithms that should then be translated into a language of the candidate's choice and also to describe an algorithm that the candidate is required to analyse and explain.

In 2013, a new qualification sequence was launched in Scotland with a major learning outcome to explain code. Students are required to explain how programs work, including *"reading and explaining code"* and *"describing the purpose of a range of programming constructs and how they work"*. Given this learning outcome, the examination authority adopted a formally defined language for use in code explanation questions, developed collaboratively between it and the authors of this paper. In truth, this language is a *reference* language with a full definition, but the examinations authority dubbed it, crucially, a *formally-defined pseudo-code*, to maintain consistency with past practice. The old and the new qualifications enable an evaluation of code written for exams, both with and without access to a formalised definition.

Evaluating questions from the old qualification scheme, where there was no formal pseudo-code specification, enables a determination as to whether a traditional view of pseudo-code as a blend of formal and natural language was prevalent among the question authors. If the traditional view is apparent in the old qualifications' questions, then the evaluation of the new scheme's questions will determine whether the "institutional memory" of the traditional informal view of pseudo-code has carried over into the new questions despite the introduction of a formally-defined language for use in assessing program comprehension.

6.2 Examination Papers Under Study

Papers from this examination authority over these two periods were evaluated to answer the two research questions. Referred to here as the *old* and *new* phases, exam papers exist for both 16 and 17 year old age groups. Exam papers are also accessible for the old phase of the 18 year old age group, but not yet in the new phase. The numbers of relevant questions available in each paper are given in Table 1, with a total sample size of 49 questions.

Table 1. Question distribution across papers analysed

Year	Old Phase			New Phase	
	16	17	18	16	17
2009	3	1	2	–	–
2010	1	2	2	–	–
2011	2	1	2	–	–
2012	1	1	2	–	–
2013	2	1	2	–	–
2014	–	–		6,8,3,3	4

One paper for each year and age group is available for analysis in the old phase. The new phase papers consist of one Specimen Question Paper for each age group, prepared by the examination authorities so that teachers have an expectation of what will appear in the exams to be sat by their students. At the time of writing, students have not yet taken examinations for the new phase. The new phase qualification for 16 year olds has had 3 further practice papers independently published but endorsed by the qualifications authority, and these have been included in the study, since they provide further evidence of the way pseudo-code is used by exam question setters. Hence the four question counts in the 2014, New Phase, age 16 cell in the table.

6.3 Evaluation Protocol

In order to address the first question that was set at the end of Section 5, the old and new phase questions, constructed without and with recourse to a formal definition respectively, were examined to determine the number of different ways that the same core construct was represented in pseudo-code. The core constructs searched for are as follows:

- Fixed repetition
- Conditional repetition
- Iteration over a data collection
- Selection with a single branch
- Selection with two branches
- Selection with multiple branches
- Assignment
- Input
- Output
- Getting the length of a list or array
- Array index

As an example,

```
<x> = <y>
```

and

```
set <x> to <y>
```

are two representations of assignment found in different questions.

Counts of different representations for each core construct were recorded separately for each of the old and new phases.

The old phase questions were also examined to determine the number of times that formal and natural language descriptions were merged in such a way that the demarcation between them was not clear, and also when knowledge had to be inferred from the question context or guessed in order to understand the pseudo-code at the Block Model's Structural level.

For example, consider the following question and pseudo-code fragments taken from a paper:

Each contestant in a game show must compete in five events. A program has been created to calculate the total … for each contestant:

```
loop 5 times
   get event points
   add points to total
end loop
```

The reader is required to guess that *event points* and *points* are in fact the same variable. Furthermore, "add X to Y" is a non-standard idiom, not used in any other paper.

The new phase questions, making use of the formally-defined reference language, were assessed to determine how often the question setters had reverted to using traditional pseudo-code techniques, where descriptive 'code' is used rather than adhering to the language definition properly, or where they had tried to exercise formality but had simply written incorrect code with respect to the definition. An example of the former is

```
start beeping noise
```

on a line of its own. While this is a functional description of what is required at this point in the program, it does not follow the language definition, it is simply an English sentence.

A simple example of the latter type of error is seen in assignment, when the following is used:

```
SET x = 3
```

instead of the syntax defined for the language, which is:

```
SET x TO 3
```

6.4 Results

The count of different representations of core programming constructs in the 25 questions contained in the 15 papers from the old phase is shown in Figure 1. For example, the first bar on the chart shows that three different ways of expressing a fixed repetition construct were found. Some examples of the differing representations used are as follows. For array indexing,

```
X[ y ]
```
and
```
x( y )
```

were seen – differing on the style of bracket used.

In some cases the differences were simply based on the case used for keywords, for example another version of assignment is

```
SET <x> TO <y>
```

while at other times it was more extreme – with this version of assignment incorporating an increment also

```
add <x> to <y>
```

Examples of the five different versions for output are

```
write <x>, <y>, …
print <x>
Display two blank lines
Display <x> and <y>
Display <text literal but no quotes>
```

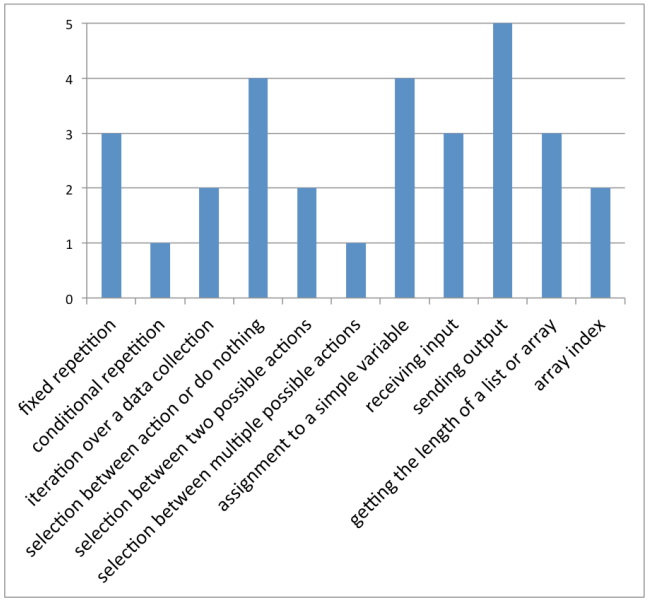

Figure 1: Count of representations of core constructs in old phase questions

When considering the new phase questions, there was a much lower variation in the number of representations used, as would be expected with a defined language as part of the qualification specification. Note however that only five papers were considered, and across the 24 questions considered, six variations were discovered across five categories: input, output, string concatenation, assignment and subprogram calling. For example, the specified form for string concatenation is

```
<str1> & <str2>
```

but this was seen several times as

```
[ <str1>, <str2>, <str3>, … ]
```

Two forms of sending to an output device were noted:

```
SEND <x> TO DISPLAY
```
and
```
DISPLAY <x>
```

and three forms of assignment

```
SET <x> TO <y>     SET <x> = <y     Let <x> = <y>
```

The subprogram calling mechanism, for example used here to turn a boiler on

```
setBoilerTo( "on" )
```

was replaced with a re-purposed version of the output construct:

```
SEND on TO boiler
```

In reviewing the 25 old phase questions, 15 of the questions contained pseudo-code where formal and natural language descriptions were merged. For example, a question about an office block with multiple floors and rooms contained the following code:

```
For each of 38 floors
     For each of 25 rooms
              Display "Floor Number:" and floor_no
              Display "Room Number:" and room_no
     Next room
     Display two blank lines
```

```
Next floor
```

The loop headers and the Display statements merge formal elements (For, Display) with natural language elements ("each of 38 floors", "and floor_no" and "two blank lines"). "floor_no" is admittedly in a formal format, using the underscore, but it is not clear where it has come from and must be inferred from context.

Considering the code contained in the 24 questions in the new phase papers, 18 of them did not meet the formal specification. While many of these were simple syntax errors, examples of natural language use occurred in four different categories, as follows.

The first category is the re-appropriation of the SEND … TO … output construct, as noted earlier, for a kind of natural language API call. This occurred in questions with external devices of various kinds. Examples are:

```
SEND Open TO lock
SEND Sound TO speakers
SEND off TO boiler
```

None of the entities used within these statements were defined in the question preamble, leaving the candidate to infer the meaning from context.

The second category again involved external devices, this time acting as input devices. The issue is the reuse of a single device name when differing devices are clearly intended. For example, in one question, a car braking system has sensors that determine the current speed and the distance to the car in front, although as this is not explicitly stated in the question it can only be inferred from the following code:

```
RECEIVE speed_of_car FROM (real) SENSOR
RECEIVE distance_to_car FROM (real) SENSOR
```

Note that the same device, SENSOR, has been used to retrieve both values. In a question that is asking about errors, and where this is not the error intended to be found by the question setter (determined from the model answer), this lack of precision will cause confusion.

The third category concerns the use of natural language embedded in the midst of well-formed statements. Examples are:

```
proceed to user screen
SEND apply brakes TO car brakes
SEND appropriate message TO DISPLAY
```

The first is pure natural language, the second another example of the re-appropriation of the output statement, only using natural language to describe what is to be sent and to where.

Finally, in a question about a central heating controller, a variable and appropriate values for an on/off switch for the controller were adopted but not defined anywhere:

```
REPEAT
   <controller code removed for this example>
UNTIL switch = off
```

6.5 Discussion

The main purpose of this small study is to determine whether practices adopted over many years using informal pseudo-code in teaching and assessment settings carry across to settings where a formally defined language is to be used for assessing PC skills. Given the paper's emphasis on highlighting the difference between how texts in formal and natural languages are comprehended, it will be an educational own-goal if formally-defined languages are introduced to assessment contexts but then still used in a manner that blurs the distinction between formal and natural language elements.

It appears from the results that core concepts were represented in a range of ways before a defined language was introduced, with 9 of the 11 core constructs used in the questions having between 2 and 5 alternatives. This is to be expected given the general understanding of pseudo-code as an informal language. Of more concern is that variation was uncovered in five categories of language construct, even now a formal specification should be used.

This suggests that question setters are still happy to develop their own pseudo-code formats or mould them from example texts or prior experience. Overall, then, in answer to question 1 in Section 5, core constructs are not being presented consistently , whether a formally-defined language, or informal pseudo-code is specified for use.

60% of the questions written using informal pseudo-code blended the use of formal and natural language in ways that were not deemed to be clearly demarcated. Again, this is the accepted mode of use for pseudo-code, particularly when used as a group-planning tool, where any ambiguity can be resolved in discussion. This resolution is quite impossible in a national exam context. Hence, the second question of Section 5, as to whether code is presented unambiguously, with clear demarcation between formal and natural languages, is also answered negatively.

The readers of this paper will most likely be experienced programmers and may not perceive that the reported issues with language use in exam settings are of any great concern. *We* will be able to infer exactly what is meant from the vast majority of these questions. But we are *not* the target audience, and the use of inference should not lie at the heart of program comprehension. The argument in this paper is that educators should, at all times, reinforce the formal nature of programming languages, and the consequent comprehension strategies that this entails. This is further supported by Soloway's findings on the importance of idiomatic uses to aid PC.

Do these findings suggest that using a defined pseudo-code / reference language in national exams should be discouraged? The reasons for adopting the approach, given earlier, still hold – that the preferences for particular programming languages among teachers militate against enforcing a single language. The root issue seems more likely to be the use of the term *pseudo-code* with all its historical baggage of informality. Strongly representing it to exam setters as a *reference language* would signpost a break from the past, and the reference language used for examination purposes in this study is outlined in the next section. Tew's work on concept inventories for introductory programming concepts [20] showed that students had little difficulty transferring their knowledge of programming from the language of instruction to the formal reference language used in the inventory, allaying any fears of transferability. Furthermore, making the research on PC widely available to schools, and stressing the educational value of distinguishing formal from natural languages, is an imperative.

7. A REFERENCE LANGUAGE WITH EXPLICIT CODE AND (NOT CODE)

The defined pseudo-code used in the new phase of the Scottish qualifications was designed principally by Michaelson and Cutts and is available at [14]. The language was defined primarily to address the paradox of examining candidates in program comprehension in the absence of a single specified language of instruction. It was termed pseudo-code for reasons of continuity with the practices of the old phase, but the findings here show that

it should be termed a *reference language*, which in reality it is. Indeed, the Scottish qualifications authority has now termed it a reference language as a result of this research.

The details of its syntax and semantics are not important, in that they are designed only to reflect syntactic conventions in pseudo-code developed over the years by question setters, avoiding a loss of continuity. Crucially, however, it *does* have a well-defined syntax and semantics, defined in sufficient detail to allow reference implementations to exist.

There is one major departure from a simple programming language: the language has an explicit *elision* construct and thus, in general, is not an executable language. This takes the form of any text within angle brackets; it is not a comment, however, and must be used in place of well-defined syntactic constructs, either a command or expression. Its purpose is to provide explicit demarcation between the rigorous programming language context, and the non-rigorous natural language context, thus allowing these styles to be explicitly mixed without confusing the reader, or undermining a developing understanding of rigour in programming.

The following example (from one of the specimen papers) shows why such a paradigm is useful. The purpose of the question is to test the candidate's ability to use a Boolean variable: the candidate is asked to change the type of the variable *check* to Boolean and adjust the rest of the code accordingly. The pseudo-code used is:

```
SET check TO 0
SET counter TO 1
RECEIVE registration FROM KEYBOARD
REPEAT
    IF cars[counter] = registration THEN
        SET check TO 1
    END IF
    SET counter TO counter + 1
UNTIL check = 1 OR counter = 101
```

This is a perfectly reasonable use of pseudo-code: the intended meaning of this fragment is clear to any competent programmer. However the code contains much that is not fully defined (eg the type of *registration*), requires the candidate to understand implicit semantics that are not necessary for the question (eg that the array *cars* is 100 items in size), and uses conventions that apply to this question alone, potentially confusing candidates who study many questions and try to derive a meaning for the whole language (eg that arrays are addressed from 1, whereas other questions assume array addressing from 0.) All of these issues are potentially harmful to the learner who may be struggling with the essential concepts of program comprehension, and also distract from the purpose of the question.

The example code can be re-written as follows, without changing the intent of the question, but dropping the undesirable effects:

```
SET check TO 0
SET reg TO <the registration number to be checked>
REPEAT
    <get the next car from the data store>
    IF <next car has registration number reg> THEN
        SET check TO 1
    END IF
UNTIL check = 1 OR <all cars have been checked>
```

The different impact is very clear; the simple use of explicit elision in place of both commands and expressions shows very clearly the demarcation of that part of the code that is fully rigorous, and that part which is not and can therefore harmlessly be expressed in natural language.

An early implementation of this language, developed by Connor, is available[1]. Code containing elisions can currently only be checked for context-free syntactic correctness, although allowing more sophisticated checking and execution for certain classes of elision is an interesting issue, currently being investigated further. However, here lies a further example of the value of a fully defined reference language: as noted, the majority of the new phase pseudo-code contained unintentional errors in both syntax and semantics, avoidable were a reference implementation used.

8. CONCLUSION

In the context of the generally understood importance of program comprehension, we have elaborated on one of the most essential differences between natural and artificial languages, in terms of the context in which they must be understood. Understanding this difference is a key task in the teaching of programming, instilling the understanding that programming languages supply precisely defined unambiguous meaning derived from their internal structures only, rather than through the use of intelligence and knowledge of context required to interpret natural languages.

This distinction is crucial in educational regimes where examination of program comprehension cannot be based upon a single language. In this case, the common solution is to adopt a more or less defined pseudo-code for use in examination questions.

Analysis of such examples however demonstrates that the lack of rigor implied by the use of pseudo-code, even when the pseudo-code language is fully-defined, leads to a harmful mixing of the informal and formal languages which we strive to distinguish in our teaching. This will lead not only to poorly defined examination questions, but endangers the whole essence of teaching and assessing program comprehension.

Finally, we suggest the replacement of the use of a pseudo-code language by a well-defined reference model for use in examination. We show how the use of an explicit demarcation construction within such a language can still allow partial definitions to be valuably used in examples, but without compromising the essential principles of rigorous comprehension.

9. ACKNOWLEDGEMENTS

We thank David Bethune and Derek Middleton of the Scottish Qualifications Authority for recognising the importance of this area of study and for their significant contribution to it. We thank the many Scottish school teachers whose reports on using the reference language have extended our understanding of the area.

10. REFERENCES

[1] AQA exam board. *http://aqa.org.uk*. Last accessed. 6-10-2014.

[2] T. Bell, P. Andreae, and L. Lambert. Computer Science in New Zealand High Schools. *Conferences in Research and Practice in Information Technology*, 103, January 2010.

[3] J. Borstler, M. Hall, M. Nordstrom, J. Paterson, K. Sanders, C. Schulte, and L. Thomas. An evaluation of object oriented example programs in introductory programming textbooks. *SIGCSE Bulletin*, 41(4), January 2010.

[4] College Board. http://www.college.board. Last accessed 13-7-2014.

[5] Q. Cutts, S. Esper, and B. Simon. Computing as the 4th "R": a general education approach to computing education. In *ICER'11 Conference Proceedings*, August 2014.

[1] http://bit.ly/haggis4sqa

[6] V. Fix, S. Wiedenbeck, and J. Scholtz. Mental representations of programs by novices and experts. In *INTERCHI '93 Conference Proceeding,* April 1993.

[7] R. Lister. Concrete and other Neo-Piagetian forms of reasoning in the novice programmer. *Conferences in Research and Practice in Information Technology* 114, 2011.

[8] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. In *ICER'08 Conference Proceedings*, September 2008.

[9] J. Meyer and R. Land. Threshold concepts and troublesome knowledge – linkages to ways of thinking and practising. In *Improving student learning – ten years on*, C. Rust (ed). Oxford Centre for Staff and Learning Development, Oxford, 2003.

[10] R. Pea. Language-independent conceptual "bugs" in novice programming. *J. Educ Comput. Res.* 2(1), 1986.

[11] A. Robins. Learning edge momentum: A new account of outcomes in CS1. *Comp. Sci. Ed*., 20(1), 2010.

[12] C. Schulte. Block model – an educational model of program comprehension as a tool for a scholarly approach to teaching. In *ICER'08 Conference Proceedings*, September 2008.

[13] C. Schulte, T. Busjahn, T. Clear, J. Paterson, and A Takerkhani. An introduction to program comprehension for computer science educators. ITICSE'10 Working Group Report, June 2010.

[14] Scottish Qualifications Authority exam board. *http://sqa.org.uk*. Last accessed. 6-10-2014.

[15] P. Siebel. Code is not literature. *http://gigamonkeys.com/code-reading*. Last accessed 6-10-2014.

[16] J. Sorva. Notional machines and introductory programming education. *ACM Trans. Comput. Educ*. 13(2), June 2013.

[17] E. Soloway, J. Bonar. and K. Ehrlich. Cognitive Strategies and Looping Constructs: an Empirical Study. *CACM 26(11), November 1983.*

[18] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Soft. Eng. 10(5), September 1984.*

[19] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *CACM* 29(9), September 1986.

[20] A. Tew. Assessing fundamental introductory computing concept knowledge in a language independent manner. Doctoral Thesis. Georgia Institute of Technology, December 2010.