Intervention tools for low end programmers

Dale Parsons
Otago Polytechnic
Dunedin, NZ
dale@tekotago.ac.nz

Abstract

The aim of this study was the development of tools to pin point the programming problems experienced by the bottom 25% of students. Three tools were developed and are described in response to the results from a standard coding test.

Tool 1: Given a programming problem to code - usually a simple game or scenario that requires a loop with a nested decision - can the students pick what the structures are?

Tool 2: Given code but not a description of the problem, can they describe how the program will run? Once it has been demonstrated, can they draw an activity diagram that matches the code? In the electronic version can they desk check each variable?

Tool 3: Given a description of the game and a list of all the lines of code but jumbled, plus some extra structures, can they drag and drop the lines into a workable order?

It is hoped that these tools will diagnose problems and act as intervention tools before the students sit a standard test: where they plan, draw an activity diagram and code a given problem.

Keywords: novice programmer, intervention

1. Introduction

An area that is of much interest to many others in Computing Education is the development of tools to help people learn programming (Parsons and Haden 2007).

We have previously described the use of automated puzzle-style exercises we are able to provide a rote learning tool that addresses specific syntactic features, models good programming practice, and is, according to early student feedback, even fun (Parsons and Haden 2006).

In this paper we describe the development of tools to pin point the programming problems experienced by the bottom 25% of students. Three tools were developed and are described in response to the results from a standard coding test.

This Supplementary Proceedings paper appeared at the *21st Annual Conference of the National Advisory Committee on Computing Qualifications* (NACCQ 2008), Auckland, New Zealand. Samuel Mann and Mike Lopez (Eds).

In recent years, computer programming has become a common and accepted part of the undergraduate curriculum. There has been, and continues to be, vigorous exploration and debate within the Computer Science Education community about the best programming paradigm (Decker, R. & Hirshfield, S., 1994; Pugh,J.R.,LaLonde, W.R., & Thomas, D.A., 1987; Bergin, J.,2000), language (Allen, 1998; Duke, R.. Salzman, E., Burmeister, J., Poon, J & Murray,L, 2000), programming environment (Hu, M., 2004) and philosophical approach (Stein, L.A, 1998; Fincher, S. 1999) to the teaching of Introductory Programming. The content of a first programming course must necessarily focus upon syntactic and mechanical learning to a certain extent.

The focus in this paper is on techniques to identify problems with learning programming, not just in the so-called "novice programmer", but in the weakest 25% of that group – the "low end programmer".

For this group, the problems of programming become amplified. Learning syntax is not just a hurdle but a complex barrier. Without a good grasp of the syntactic rules of a language, programs cannot be written or perhaps even comprehended.

Many educators have identified the importance of task engagement in learning (Kearsley & Shneiderman, 1999), and this issue must be addressed if we are to use drill techniques to teach programming language syntax successfully. A second problem with using drill techniques in programming is the difficulty of removing a single syntactic unit from the logical context in which it occurs.

We can easily isolate the computational act of multiplication from the conceptual exercise of solving a complex word problem, but it is not so clear how to separate, for example, "correct placement of semi-colons" from a complex programming task in which they might be used. Beginning programmers who are set a programming exercise can easily become lost in the logic of the solution, and end up with little opportunity to practice correct placement of semi-colons.

We have previously described "Parsons Programming Puzzles", a set of drag-and-drop style exercises designed to provide students with the opportunity for rote learning of syntactic constructs in Delphi. The exercises were designed under the following principles:

- Maximizing engagement

- Constrain the logic
- Permit common errors
- Model good code
- Provide immediate feedback

In this paper we describe the use of automated tools for use in a first programming class as intervention tools for low end programmers. These early intervention tools are intended for use before the first formal assessment.

The first assessment in our introductory programming course is a standard coding test where the students must diagram and code a simple game. It is an excellent diagnostic tool for describing what students can and can not do. Unfortunately for the bottom 25% of students it often comes too late in the course for successful interventions. Often the worst programmer does so little in the test it is very hard to pin-point their problems.

Using the problem descriptions laid out by Garner, Haden & Robins as a rough guide we categorized the problems experienced by the bottom 25% of students in their first test in Table 1.

Table 1: Problems experienced by the bottom 25% of students in test 1

	student								
Problem	1	2	3	4	5	6	7	8	9
Understanding the task	x		x						
Game plan/ logical steps	x		x						
Use of variables			x			x			x
Expressions and calculations		x		?	x		x		
Loops	x	x	x	x	x	x		x	x
Selection	x		x	x	x	x	x		x
Nested structures				?	?	x		x	?

(?: means the student has not written enough code to be able to assess this category)

2. Tool One

The biggest problem is where students do not understand the task, assuming that this is not a literacy problem then what they can not do is select which programming structures are needed. Tool one is a description of a game or a multi-structured task that usually involves a loop and a nested decision, the students must select the most appropriate type of loop and selection construct (Figure 1).

This is a fill in the gaps type of quiz, the clue button [?] will show the line in the description that is describing the structure. In the above example the outer structure is a repeat loop and the line in the description that indicates this is "the user is asked if they want another game". I have been using a non electronic version of these questions for several years but have noticed that the weakest students don't attempt to answer them in class they wait until the answers are being given out and copy them down, the quiz format means they have to attempt the questions before they can have the answer revealed to them. I have made up 6 of these questions.

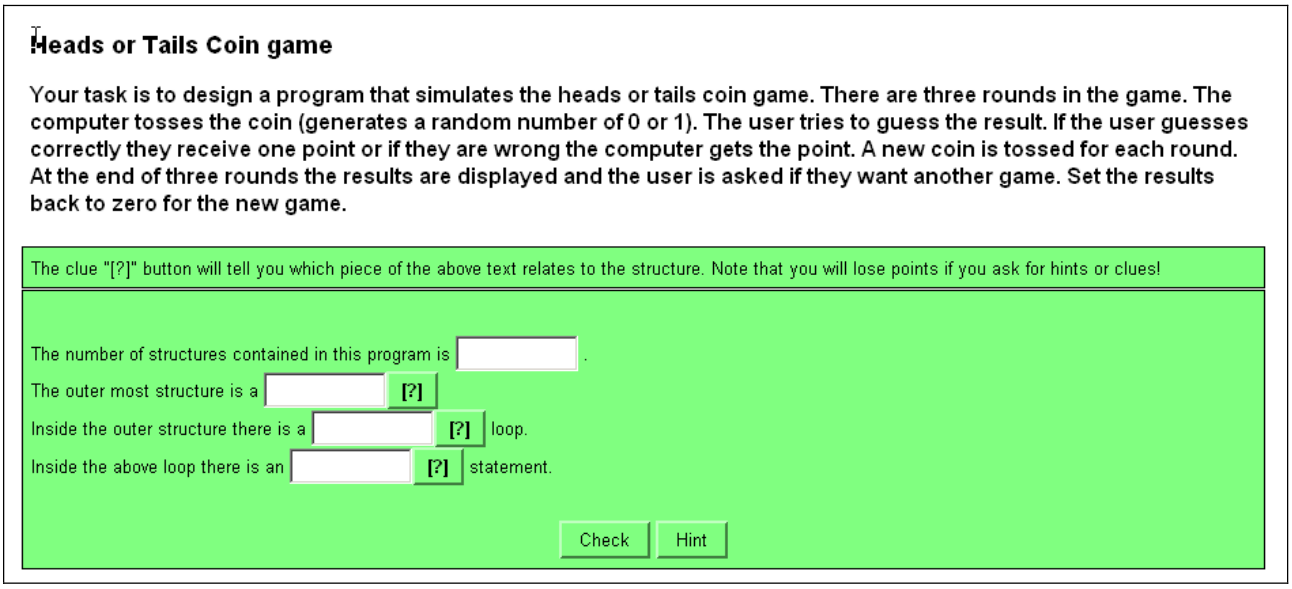

Figure 1: Tool One

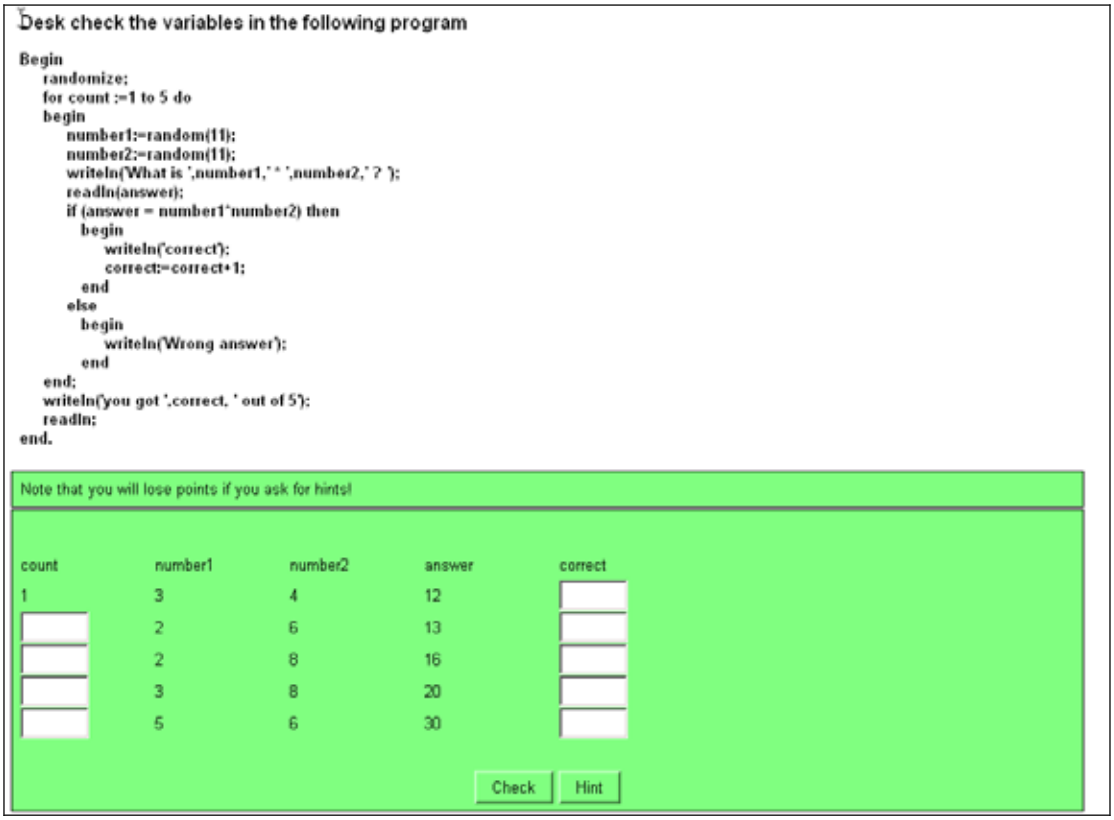

Figure 2: Tool 2

3. Tool 2

This tool was an attempt to address the problem of game plan, program flow, constructs and variable use (Figure 2). Could they read and understand code that they had not written? Given a programming problem could they describe how the program runs? They can check their description by running the problem. Can they draw an activity diagram that matches the code? This highlights where the nested structures are and gives them a different visual view of the code. The electronic tool that matches each problem gets them to fill in a table, desk checking each variable as if the game or program was being played or run. Filling in the desk check gets the students working with well named variables and performing the calculations, where the program randomly generates a variable's value or the user is asked for a value, I have hard coded that value into the table.

4. Tool 3

This tool focuses on the structure of the code avoiding the small syntaxic errors that can stall a beginning programmer (Figure 3). Given a description of the program and all the lines of code but jumbled plus some extra distracter lines of code, can they drag and drop the lines into a workable order? In the first version of this tool I took out the ability to check their answers and any indenting code clues. The students had to rearrange the code and when they were satisfied with their solution they printed out the code.

This tool was trialled with the students from three classes. For two of the classes I collected in the printouts but did not look at them until after the test was sat and marked, then I compared them with their test to see whether they were a good diagnostic tool of their programming ability. The third class which is a tiny stream of design students got an intervention straight away as I noticed that six out the seven members of the class had all incorrectly placed the repeat statement in their program straight above the until statement (the repeat line should have been at the beginning of the program as it was the outer loop). Of those six students four used a repeat statement correctly in their test, one made exactly the same error again and the other correctly placed the repeat but forgot the word until in the condition line.

This tool showed up huge deficiencies in the bottom 25% of students. Oddly one student dragged and dropped the lines of code in a perfect order but was unable to code a loop or a case statement in his test. Those that did not understand the task and the steps required in the test program also could not get the drag and drop lines anywhere near a workable order.

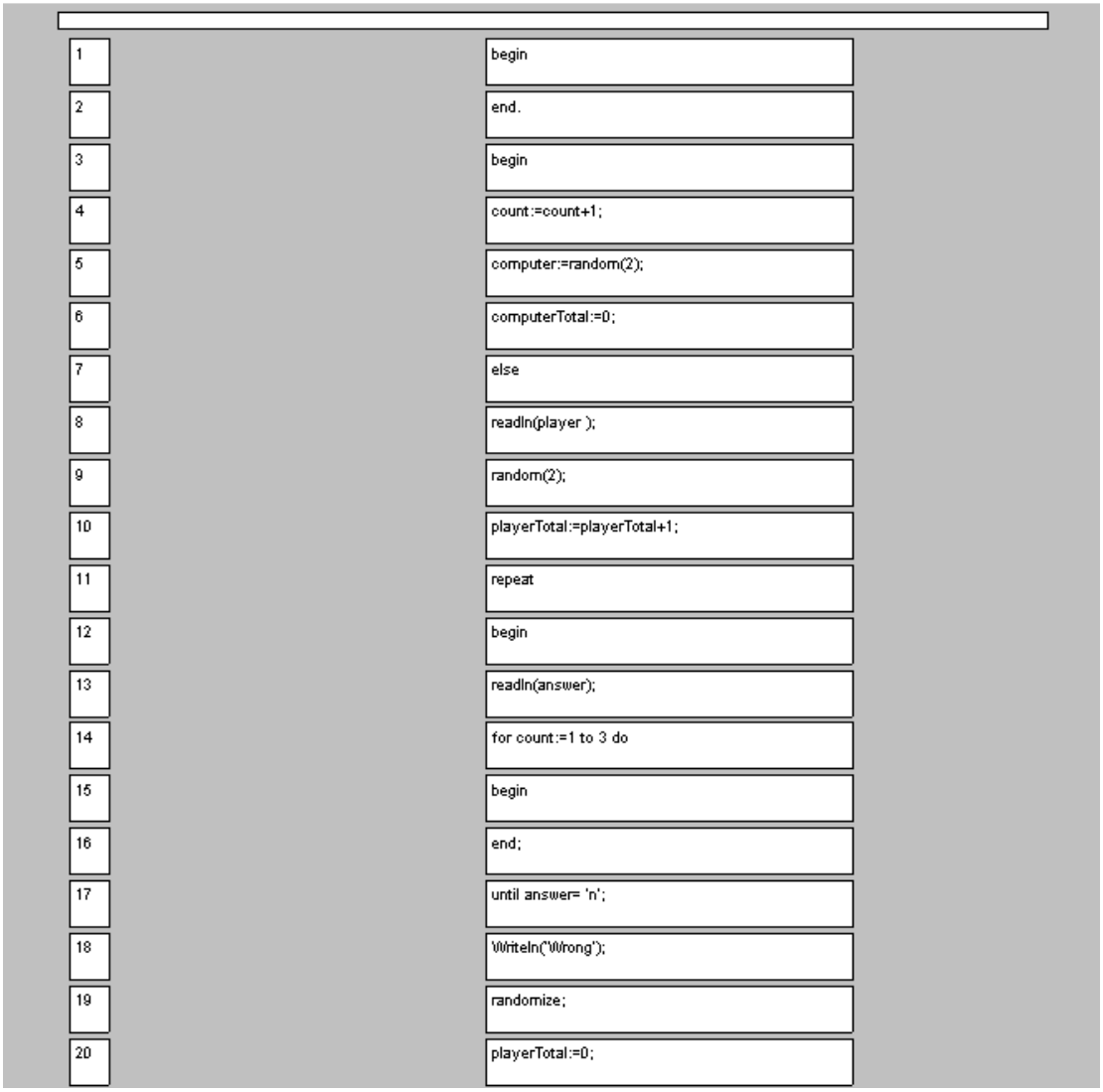

Figure 3: Tool 3

5. Development

Given that this tool seems a very good predictor for success in the test I have turned them into the programming puzzles described in Parsons and Haden 2006. They are now fully automated so that the students can check their work and keep rearranging the lines until they get them all in the correct order. This saves the Lecturer having to mark each puzzle attempt but it relies on the student reflecting on their errors, hopefully remembering bits of the structural syntax and not randomly dragging and dropping the lines until they are correct.

As they are now self checking the students need to be able to distinguish between the four begin lines so the indenting clues are necessary. In the first version where I was observing the results, I thought the indenting clues would make the puzzle too easy but clearly for the bottom 25% the activity was too difficult. These tools are not aimed at the good programmers they should just code up their solutions and let the compiler give them feedback on their errors, this is for the student that can't get started or they get started but with a bad plan.

Further research will provide statistical evaluation of these tools.

References

Allen, R.K., Bluff, K., Oppenheim, A. (1998) Jumping into Java: Object-Oriented Software Development for the Masses. *ACSE 1998*, Brisbane.

Bergin, J., (2000) Why Procedural is the Wrong First Paradigm if OOP is the Goal, website,csis.pace.edu/~bergin/papers/WhyNotProceduralFirst.html,

Booch, G., Rumbaugh, J. & Jacobson, I., (1999), The Unified Modeling Language User Guide, Addison Wesley, Reading, Massachusetts.

Decker, R. & Hirshfield, S. (1994) The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS1, *SIGCSE 94*

Duke, R.. Salzman, E., Burmeister, J., Poon, J., Murray,L., (2000) Teaching Programming to Beginners – choosing the language is just the first step, *Proceedings of the Australasian conference on Computing education,* pp. 79-86., December 2000, Melbourne Australia.

Fincher, S., (1999) What are we doing when we teach programming?, In *Frontiers in Education '99*, pages 12a41-5. IEEE, November 1999.

Grant, F. & O'Brien, H., *Maths Plus for New Zealand Schools*, (2000) Horowitz Martin, St. Leonards, Australia.

Hu, Minjie (2004), "Teaching Novices Programming with Core Language and Dynamic Visualisation", *Proceedings of the NACCQ 2004*, Christchurch, New Zealand, 6-9 July, 2004, pp. 95-104.

Kearsley, G. Sheniderman, B., (1999) Engagement Theory: A framework for technology-based teaching and learning, http://home.sprynet.com/~gkearsley/engage.htm

Norman, D. A. & Spohrer, J., C. (1996), Learner-Centered Education, *Communications of the ACM*, 39, 4, pp 24-27.

Parsons, D., & Haden, P. (2006). Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In Tolhurst, D & Mann, S. Proceedings of the Australasian Computing Education Conference 2006, Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 52,

Parsons, D. and P. Haden (2007). Programming Osmosis: Knowledge Transfer from Imperative to Visual Programming Environments. *20th Annual Conference of the National Advisory Committee on Computing Qualifications*, Nelson, New Zealand, NACCQ in cooperation with ACM SIGCSE.

Stein, Lynn Andrea (1998) What We've Swept Under the Rug: Radically Rethinking CS1. *Computer Science Education* 8(2):pp.118-129.