Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses

Dale Parsons
School of Information Technology and Electrotechnology
Otago Polytechnic
Dunedin, New Zealand
dale@tekotago.ac.nz

Patricia Haden
School of Information Technology and Electrotechnology
Otago Polytechnic
Dunedin, New Zealand
phaden@tekotago.ac.nz

Abstract

Mastery of basic syntactic and logical constructs is an essential part of learning to program. Unfortunately, practice exercises for programming basics can be very tedious, making it difficult to motivate students. In this paper we describe Parson's Programming Puzzles, an automated, interactive tool that provides practice with basic programming principles in an entertaining puzzle-like format. Careful design of the items in the puzzles allows the tutor to highlight particular topics and common programming errors. Since each puzzle solution is a complete sample of well-written code, use of the tool exposes students to good programming practice. This paper discusses the motivation of Parson's Programming Puzzles, and gives several examples. We describe the web-based authoring tool used to build the puzzles, and present our plans for future development.

Keywords: Teaching introductory programming; interactive teaching tools.

1 Introduction

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. While these issues are critical, we can also gain insight from recognising what the teaching of programming shares with the teaching of other disciplines. Learning to program is in some ways similar to learning a new spoken language. In both cases one must master syntax (e.g. placement of semi-colons and keywords; specification of the bounds of a for-loop) and semantics (e.g. algorithm design and class architecture). The acquisition of a programming language also bears some resemblance to the early learning of mathematics, where we can make the same distinction between syntactic or mechanical issues (e.g. the multiplication tables) and the development of abstract thinking skills (e.g. solving of complex word problems).

The content of a first programming course must necessarily focus upon syntactic and mechanical learning to a certain extent. Without a good grasp of the syntactic rules of a language, programs cannot be written. The comparable material in maths or foreign language courses has traditionally been taught via "rote learning" techniques, repeated exposure through practice and drill (Norman, D. A. & Spohrer, J., C. 1996; see, for example, O'Neill, 1972). The pages of long division problems that we solved as children are still used today (see, for example, Grant, F. & O'Brien, H., 2000), and even game-based computer programs for teaching early mathematics still rely on drill to a significant extent (see, for example Treasure Mountain www.broderbund.com). It is not unreasonable to assume that repetitive drill learning would also be an appropriate technique for teaching syntactic rules in a first programming course.

Unfortunately, drill exercises in any discipline suffer from the very sameness that makes them effective. They're boring. Writing out 100 for-loop headers would be just as tedious as working through a page of 100 long-division problems. Many educators have identified the importance of task engagement in learning (Kearsley, G. & Sheniderman, B, 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.

In this paper we describe an automated tool for use in a first programming class. Parsons Programming Puzzles comprises a set of drag-and-drop style exercises designed to provide students with the opportunity for rote learning of syntactic constructs in Turbo Pascal. The exercises are designed under the following principles:

Maximize the engagement: Games and puzzles have a history of appeal stretching back many centuries. The value of game-style interfaces has recently been considered by Prensky (2001), who describes their effectiveness in promoting engagement in a variety of contexts. Hall and his colleagues (2003) advocate the use of puzzles and games in CS teaching in both beginning and senior level courses. They note that this approach helps to accommodate a broader range of individual learning style than do more traditional methods. We assume that making a task "puzzle-like" will make it more engaging. This will help to insure that students will work with the tool enough to gain the repeated exposures required for effective rote learning.

Constrain the logic: As described below, Parson's Puzzles provide a good code structure (i.e. semantics) in which the student is required to make syntactic judgements. Thus the student is unable to become sidetracked by his or her inability to find the correct logical/algorithmic approach to the problem.

Permit common errors: In our teaching experience, we have seen that students tend repeatedly to make certain common syntactic errors. Parson's Puzzles intentionally allow students to make these errors by providing them as distracter options. This enables students to make a direct comparison between what they commonly do (which is wrong) and what the correct syntactic construct looks like.

Model good code: The beginning programming student may be able to produce an apparent solution to a programming task (i.e. it generates the correct output), which is actually incorrect from a design perspective. For example, in a complex branching problem, a student may use many individual if statements rather than a more concise and appropriate nested if structure. The student consequently doesn't get to see, and may even be unaware of, the good coding solution. The correct solution to each Parson's Puzzle is a fragment of good code. Because students repeat each puzzle until they have achieved 100% accuracy (see below) we insure that they have been exposed to the correct coding approach.

Provide immediate feedback: The beginning programmer can often spend a considerable amount of time struggling with a non-working program, unable to determine the precise location of the error (debugger feedback, while often useful, is not always completely diagnostic to the beginner). With Parson's Puzzles, incorrect choices are immediately identified.

2 Puzzle Structure

All Parson's Puzzles are drag-and-drop style. In each problem context, the student is given a selection of code fragments, some subset of which comprise the problem solution. The draggable fragments may contain single or multiple lines of code. The student drags their choices into the indicated answer locations. By clicking on the "Check" button, they receive feedback about which of their choices are correct, and which are not. The student is encouraged to repeat the problem until 100% accuracy is achieved. The correct solution for each puzzle is a complete small program or subroutine. Figure 1 shows a typical puzzle in its initial state.

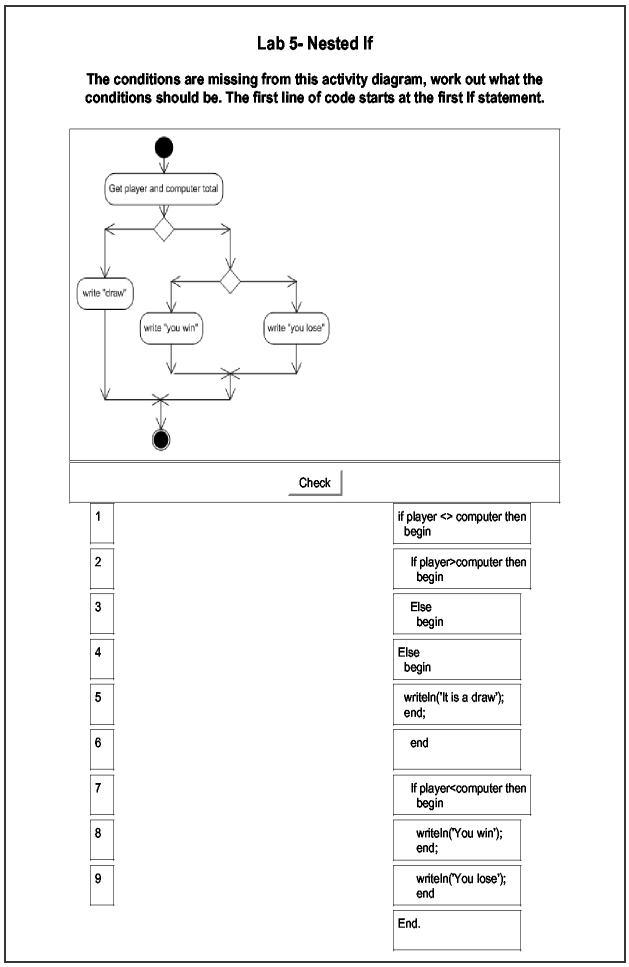

Figure 1: A Typical Parson's Programming Puzzle

Each puzzle includes a description of the program to be built. This description may be simply textual (e.g. "Produce the even numbers 2..20 on the screen"), or may be given as a UML-style Activity Diagram (cf. Booch, G., Rumbaugh, J. & Jacobson, I., 1999), as shown in Figure 2.

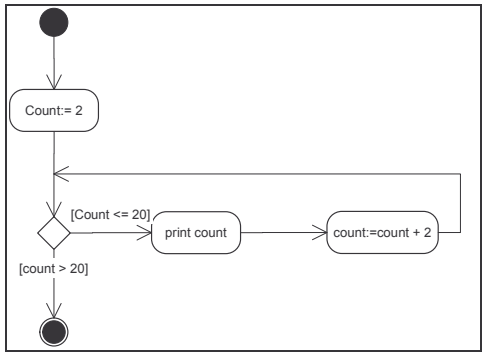

Figure 2: Activity Diagram

The use of Activity Diagrams is an important theme in Parson's Programming Puzzles. These diagrams (and other flowchart and/or pseudo-code approaches) are an important tool for good program design (Olsen, 2005; Tabrizi, M., Collins, C., Ozan, E., & Li, K., 2004). Their use encourages high-level planning, which promotes good logical structure and modularisation. When this kind of planning is done prior to the start of coding, the programming process is smoother, with fewer wrongs turns and backtracking, and the resultant code has better structure (Booch, et. al.,, 1999). In spite of these clear benefits, many students are reluctant to use Activity Diagrams, either from confusion about how to build them, or impatience to leap directly into coding. By providing Activity Diagrams as the problem description in Parson's Programming Puzzles, we model good code and a good coding process. By exposing students to many exemplars of this planning tool, we hope to make them able to produce their own diagrams in novel solutions.

The use of Activity Diagrams also gives the problem designer very fine-grained control over the material being demonstrated in each puzzle. For example, by providing a diagram from which the branching conditions have been omitted, one requires the student to figure out the correct Boolean expressions. The student is thus able to produce a logically non-trivial program while concentrating on a single syntactic construct.

3 Designing Distractors

One of the benefits of the drag-and-drop approach is that the designer constrains both the correct options, and the incorrect options. Thus we can narrow the very large space of possible errors to only those that can be used to illustrate a particular point. For example, Pascal students often have trouble learning to print single quotes (the literal string delimiter). To focus attention on this syntactic feature, an early puzzle provides the following two items:

Correct: Writeln('Don''t use a line that''s not correct');

Distractor: Writeln('Don't use a line that's not correct');

Response items can also be designed to highlight programming principles the student may ignore. The following pair of items requires the student to use a previously declared constant:

Correct: Total := TICKETPRICE * TicketCount;
Distractor: Total := 11.50 * TicketCount;

Items can be designed to compare correct keywords, formatting rules, and even proper indentation. By carefully selecting the statement(s) in each correct choice and each distractor, puzzles can be built to highlight a very wide range of syntactic principles from very simple and obvious to very subtle or complex.

4 Sample Puzzles

4.1 Order of Statements:

The most basic concept behind computer programming is the sequential order of computer instructions. However, it is not always easy for the novice programmer to identify the correct order in a complex task. The Parson's Programming Puzzle shown in Figures 3a, 3b and 3c requires the student to correctly order a series of Writeln statements to produce sensible output. Since the required statements are provided, the student cannot become distracted by the many possible syntactic errors; his or her focus is constrained to the issue of logical ordering.

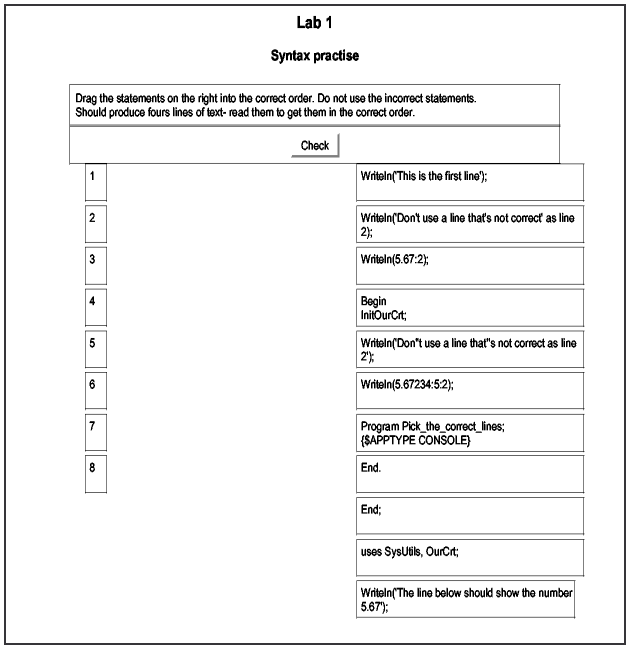

Figure 3a: Order Of Statements Puzzle

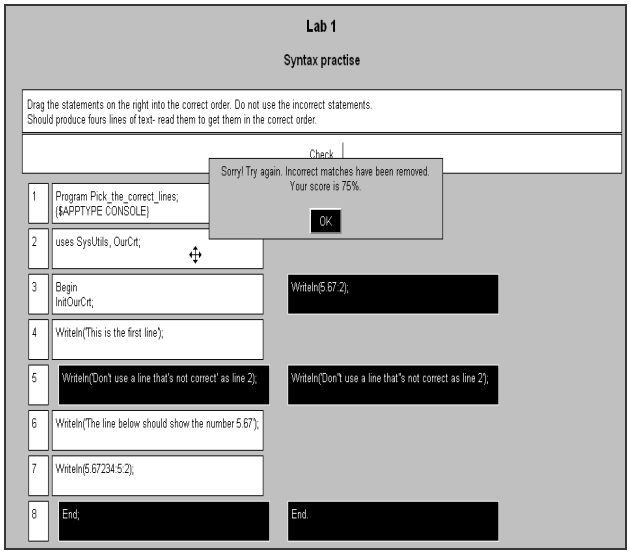

Figure 3b: Partially Correct Solution After Marking Showing Error Feedback

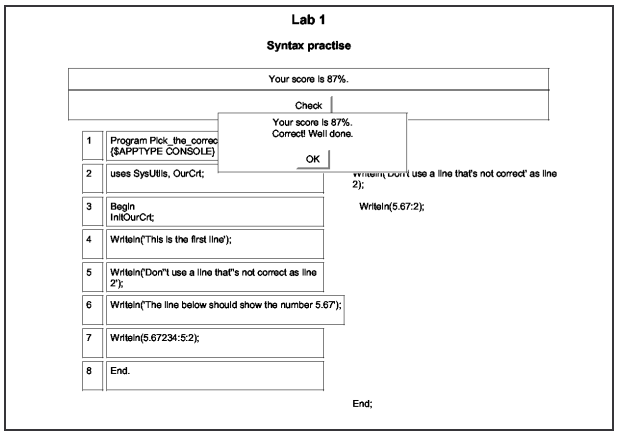

Figure 3c: Solution to Order of Statements Puzzle

4.2 Variables

Students initially have difficulty distinguishing between the contents of a variable and the name of the variable. In the Puzzle shown in Figures 4a and 4b, the student must correctly use a variable, not a literal string, to produce the program output. In addition, the student must select between a good variable name and a poor variable name in order to successfully complete this puzzle.

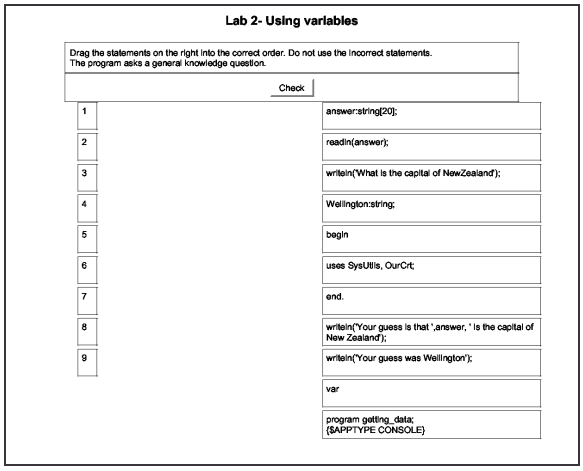

Figure 4a: Variables Puzzle

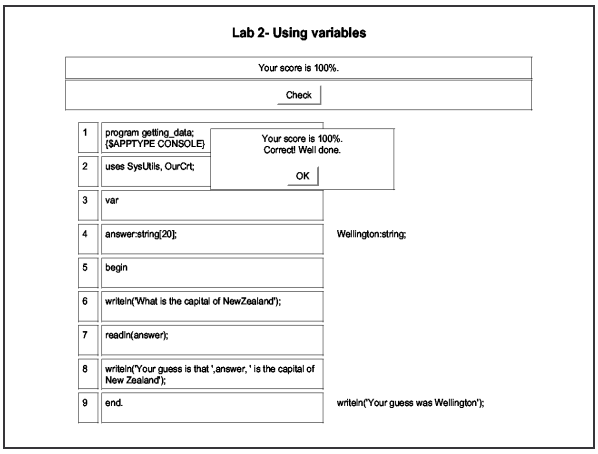

Figure 4b: Solution to Variables Puzzle

4.3 Nested For Loops

Students often have difficulty understanding both when one would use a nested for loop, and how the structure will behave. In the puzzle shown in Figures 5a and 5b, these two issues are separated. The options constrain the student to use a nested for; the student is responsible for correctly identifying which should be the inner and which the outer loop.

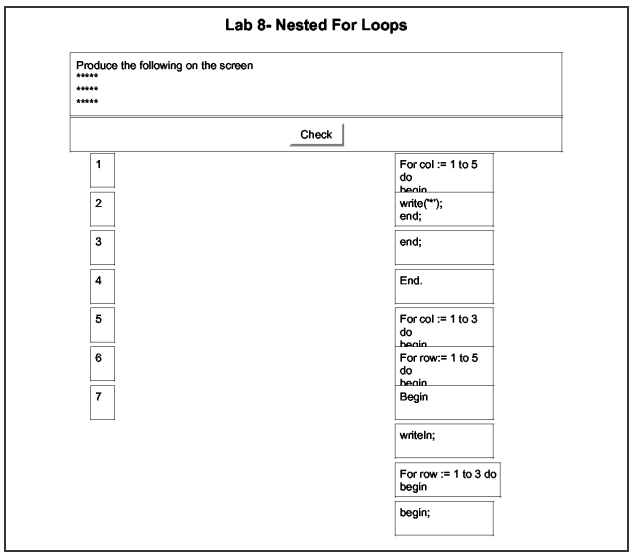

Figure 5a: Nested For Loop Puzzle

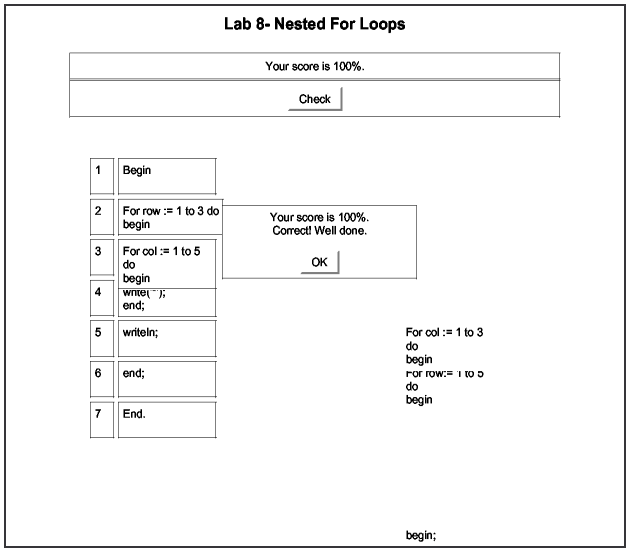

Figure 5b: Solution to Nested For Loop Puzzle

5 Implementation

Parson's Programming Puzzles is implemented as a web-based application. It can thus be used for in-class work in a networked computer lab, or be accessed independently outside of class from any web browser. It is implemented using the Hot Potatoes authoring tool (http://web.uvic.ca/hrd/halfbaked/). Hot Potatoes makes construction of drag-and-drop problems extremely simple. The designer simply provides the question header (which can contain image files) and the draggable items (correctly placed solution items, followed by distractors). At run-time, solution and distractor items are randomised.

Figure 6 shows a typical Hot Potatoes development session in progress.

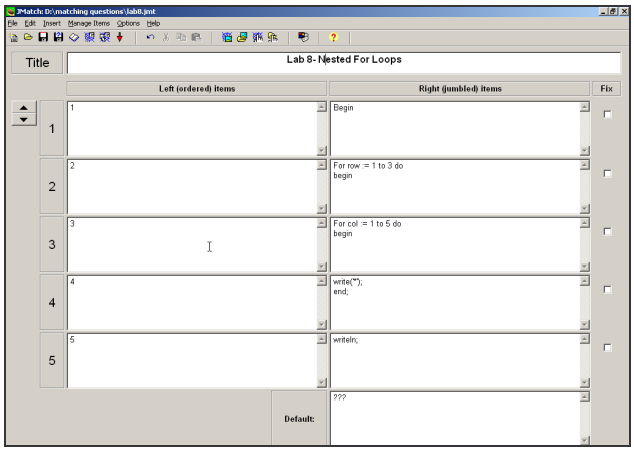

Figure 6: Using Hot Potatoes to Build a Puzzle

To deploy the application, one provides a path to a server, and can then configure other optional features such as whether an e-mail is to be generated when a problem solution is submitted. Although Parson's Programming Puzzles currently use only the drag-and-drop format, Hot Potatoes supports a number of other problem types including multiple choice, fill-the-gap and crossword, which we hope to use in the future.

6 First Experiences

Parson's Programming Puzzles have been piloted in two first programming courses at Otago Polytechnic. We initially created one puzzle for each lab topic. The puzzle was used as a revision tool, worked in-class at the start of the lab following the one where the topic was covered.

During the first trial course, the tutor observed students' performance directly to note the pattern of errors that were made, and to ensure that the level of difficulty was appropriate. We preferred that 100% accuracy be achievable in at most two or three attempts.

Puzzles took approximately five minutes to complete, indicating that they were tractable, and that, were a larger pool available, students would not find it onerous to work through many exemplars of a topic, gaining the necessary exposure for effective rote learning. First versions of some puzzles were too difficult, with students unable to achieve 100% accuracy in the desired number of tries. These puzzles were modified to reduce their difficulty. Making the distractors more distinct from the correct choices resolved this problem in some cases. For puzzles that originally had only text descriptions, adding the appropriate Activity Diagram also greatly improved performance, demonstrating the value of providing a good logical structure to inexperienced programmers.

Students did, as expected, false alarm on the "common error" distractors, thus getting an opportunity to compare their incorrect syntax with the correct statement in the problem solution. More detailed analysis of error patterns is planned in the future.

After the second trial course, students completed a questionnaire about the puzzle tool. Students were asked to rate the usefulness of the puzzles for initial learning and for revision. Students were also asked to list things that would make the puzzles more useful and things that would make the puzzles more interesting. 17 students completed the questionnaire.

Rating: The summary of student ratings is shown in Table 1.

	Not Useful	Useful	Very Useful
How useful did you find the puzzles for learning Pascal?	18%	47%	35%
How useful did you find the puzzles when revising for your test?	53%	35%	12%

Table 1: Student Ratings of Puzzle Tool. N = 17

14 of 17 respondents (82%) rated the puzzles as either Useful or Very Useful for learning. Although the sample size is small, we are nonetheless encouraged by this high positive response rate to continue to develop Parson's Programming Puzzles.

Only 8 of 17 respondents (47%) reported finding the puzzles useful for exam preparation. Based on student comments, we believe that this problem may have been partly because the puzzles were too difficult to access outside of class. We are hopeful that modifying the mode of puzzle delivery (see below) will help make them a more useful revision tool.

Comments: There was considerable overlap in student comments, clearly highlighting aspects of the puzzles that need to be improved in future versions.

When asked to "List three things that would make the puzzles more useful", the students offered the following suggestions:

- Provide more exemplars of each problem type (4 respondents)
- Provide more varieties of puzzle not just drag and drop. (5 respondents)
- Improve the "fussy" mechanics of drag and drop (6 respondents).
- Implement the puzzles as a stand-alone application rather than as a webpage (9 respondents)
- Provide more detailed feedback about what errors are made (14 respondents)

For this initial study, we prepared only one puzzle per lab topic, and all puzzles were drag and drop. Expanding the item pool and increasing the variety of puzzle formats should resolve the first three items in the list above. Clearly we must also focus on the delivery platform, and, most critically, on the type of feedback provided to the student. We discuss our plans to address these issues below.

When asked "List three things that would make the puzzles more entertaining", students made the following comments:

- Add sound (3 respondents)
- Add colour (7 respondents)
- Add animations (5 respondents)
- Provide some kind of reward (a sound or animation) when the correct solution is achieved (5 respondents)

These comments all suggest that comparatively simple changes to the program interface will make it more entertaining to use. Such features, while perhaps not directly related to the educational value of the tool, would further increase enjoyment and motivation, causing students to spend more time working with the tool.

7 Future Directions

The current pilot version of Parson's Programming Puzzles has demonstrated that puzzles can be constructed to demonstrate important syntactic principles while targeting common errors, modelling good code, and being enjoyable to use. The exercises can be easily implemented using the Hot Potatoes authoring tool. We continue to use the puzzles in our introductory programming course, and plan to develop the tool in the following ways:

Increasing the Problem Pool: We are currently building more puzzle exemplars, and hope to eventually have 50 to 100 puzzles per topic. A student working on a particular topic could then be presented with 10 to 20 puzzles at each session, randomly selected from a large pool. Students could thus use the tool many times for revision of the same topic, encountering a novel combination of problems each time.

Enhancing User Feedback: At the moment, students receive immediate feedback for each question showing the percentage of correct items, and highlighting the incorrect items. Many students requested even more detailed feedback about the errors they had made. Among the specific suggestions made were:

- Have two windows so students can compare their incorrect solution to the correct solution.
- Give immediate feedback (via sound or visual cue) when an attempt is made to place an item incorrectly.
- Show the correct solution after some fixed number of incorrect attempts.
- Provide a comment or explanation discussing the student error.

The first four suggestions are simple interface adjustments, and can be easily implemented in future versions. The final suggestion – include a discussion of the student error – is more complex. To provide very detailed feedback of this kind, it would be necessary to prepare text comments for each of the potential errors (or classes of errors) that can be made for each puzzle, and to map each possible incorrect arrangement of items to one of the prepared comments. This assumes that we can reliably identify a student's conceptual misunderstanding simply from observing the incorrect placement of distractor items. We doubt that this is possible. It is our experience that it can be challenging to localise the causes of a student's confusion even with the opportunity for extensive discussion, much less from a single drag and drop error. A practical alternative might be to provide a general explanation of the issue addressed by each puzzle (with examples of correct code) that the student could request if they are struggling with a particular puzzle.

In addition to such local feedback, it would also be useful to allow students to track their long-term progress, showing the number of errors they make for a given topic over time. This will help to identify their areas of weakness, and provide reinforcement as they master the topic.

Enhanced User Interface: Our current interface appears distinctly plain in these days of graphics-intensive educational software. The current web-based implementation also lacks flexible navigation (students must access each lab individually from a text directory listing). Many students commented on the lack of sound, animation and colour, and expressed a preference for a stand alone Windows-style application, rather than the current web-based form.

We are currently working toward implementing Parson's Programming Puzzles as a stand alone application with a more stylish interface, incorporating attractive graphics and easy navigation between topics. In response to student feedback, we also plan to include animation in a help facility, and as "rewards" for completing puzzles.

Enhanced Data Collection: The patterns of errors students make when working the programming puzzles may provide insight into areas where their conceptual models are especially inappropriate. By noting these patterns we may be able to adjust our teaching approach to address these problems. In future versions of the tool, we would like to be able to keep detailed records of which items are chosen during each iteration of the puzzle.

In addition we would like to record the amount of student usage of the tool outside of class. These data would give a measure of how enjoyable the tool was to use, and how effective the students perceive it to be. If it is possible to equate amount of practice directly to student performance, we will be able to measure directly the efficacy of this type of rote learning exercise for our students. Unfortunately, our early attempts at this sort of data collection have highlighted a potential difficulty. We recently modified the tool to require the student to give a username to begin a puzzle. When the puzzle is completed successfully, an e-mail is automatically sent to the tutor giving the username and results on the puzzle. After introducing this feature, we found that we were receiving fewer e-mails than there were students using the tool. Some students, concerned about the loss of anonymity, were exiting the tool before they reached 100% accuracy to prevent the e-mail being sent. Clearly this issue must be addressed before attempting to collect detailed usage data that identifies the student.

8 Conclusion

Many aspects of early programming education lend themselves to drill-based rote learning. To be effective, such exercises need to be made interesting and well-focussed. By using 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, useful and even fun. We plan to develop this tool further to provide a more flexible learning experience, and allow us to better measure its effectiveness.

9 References

Allen, R.K., Bluff, K. and Oppenheim, A. (1998): Jumping into Java: Object-Oriented Software Development for the Masses. *Proceedings of the 3rd Australasian conference on Computer science education*, Brisbane, Australia, **3**:165-172.

Bergin, J., (2000): Why Procedural is the Wrong First Paradigm if OOP is the Goal, http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html. Accessed 14 Oct 2005.

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

Decker, R. and Hirshfield, S. (1994): The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS1, *Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, Phoenix, Arizona, USA, 51-55*,* ACM Press.

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

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

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

Hill, J., Ray, C., Blair, J and Carver, C. (2003): Puzzles and games: addressing different learning styles in teaching operating systems concepts, *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, Nevada, USA, 182-186.

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

Kearsley, G. and Shneiderman, B., (1999): Engagement Theory: A framework for technology-based teaching and learning.

http://home.sprynet.com/~gkearsley/engage.htm. Accessed 14 Oct 2005.

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

Olsen, A.L. (2005): Using pseudocode to teach problem solving. *Journal of Computing Sciences in Colleges* **21**(2):231-236.

O'Neill, P.J. (Ed.) (1972): *The Shape of Mathematics*, Wellington, New Zealand, Reed Education

Prensky, M. (2001): *Digital Game-Based Learning*, McGraw-Hill.

Stein, Lynn Andrea (1998): What we've swept under the rug: Radically rethinking CS1. *Computer Science Education* **8**(2):118-129.

Tabrizi, M., Collins, C., Ozan, E., and Li, K., (2004): Implementation of Object-Orientation Using UML in Entry Level Software Development Courses, *Proceedings of the 5th conference on Information technology education*, Salt Lake City, UT, 128-131, ACM Press.