Epplets: A Tool for Solving Parsons Puzzles

Amruth N. Kumar
Ramapo College of New Jersey
Mahwah, NJ 07430
USA
amruth@ramapo.edu

ABSTRACT
Performance on Parsons puzzles has been found to correlate with that on code-writing exercises. Parsons puzzles are preferred by students over alternative programming tasks. In order to make Parsons puzzles widely available to students in the introductory programming course, we developed a tool that administers the puzzles in C++, Java and C#, called epplets. Our design of the tool improves upon the work done by earlier researchers in several ways: students rearrange lines of code rather than program fragments; they get credit based on the number of actions they take to reassemble the code; they get feedback that helps them fix their incorrect answer; and the tool adapts to the needs of the student. The tool runs as a Java Web application. We describe our experience using the tool for two years: how it benefited the students; the revisions made to address the feedback provided by the users; and our plans for future work. We found that practicing with the tool helped reduce the time and actions with which students solved successive puzzles.

ACM Reference format:

1 INTRODUCTION
In a Parsons puzzle [13], the student is presented with a problem statement, and the program written for it. The statements in the program are provided scrambled, along with distracters that are incorrect syntactic or semantic variants of existing lines of code. The student is asked to reassemble the statements in their correct order and delete any distracters. Parsons puzzles were designed to be an engaging way to learn programming [13].

Parsons puzzles have since been proposed for use in exams [3], since they are easier to grade than code-writing exercises, and yet, scores on Parsons puzzles correlate with scores on code-writing exercises. Researchers have found solving Parsons puzzles to be part of a hierarchy of programming skills alongside code-tracing [11]. In electronic books, students have been found to prefer solving Parsons puzzles more than other low-cognitive-load activities such as answering multiple choice questions and high-cognitive-load activities such as writing code [4]. Solving Parsons puzzles was found to take significantly less time than fixing errors in code or writing equivalent code, but resulted in the same learning performance and retention [5]. So, Parsons puzzles have been gaining popularity among educators of introductory programming.

Software to administer Parsons puzzles have been developed for Turbo Pascal [13] and Python (e.g., [1,7]). We wanted to make the puzzles more widely available to students in introductory programming courses. Therefore, we developed a Parsons puzzle tool for C++, Java and C#, three of the most widely used languages in introductory programming courses today. Our tool differs from prior Parsons puzzle solvers in several respects, including design, delivery, puzzle generation, and grading. To date, we have developed six modules: one on sequence, two on selection statements and three on loops. The tool is called epplets (short for extended Parsons puzzle problets) and is accessible online for free for educational use. Additional information about epplets can be found at epplets.org.

2 PARSONS PUZZLE TOOL DESCRIPTION
We will first describe the user interface of the tool, followed by description of its unique features vis-à-vis design, delivery, problem generation and grading.

2.1 User Interface
A snapshot of our tool is shown in Figure 1 at the end of this paper. The problem statement is displayed in instruction panel (I). The code written for this problem is presented in problem panel (P), both scrambled and unindented. The student has the following actions available for solving the puzzle:

- Drag a line of code from Problem panel (P) to solution panel (S);
- Drag a line of code from Problem panel (P) to Trash panel (T) to delete it – this is applicable to distracters in the puzzle;
- Reorder a line of code in Solution panel (S) by moving it up or down;
- Indent/outdent a line of code in Solution panel (S) to improve readability;
• Return a line from either the Solution panel (S) or Trash panel (T) back to the Problem panel (P), whereupon the line is placed back in its original scrambled order in the Problem panel (P);
• Move a line from the Solution panel (S) to the Trash panel (T) or vice versa.

Feedback is provided to the student in the feedback panel (F). When the student is solving the puzzle, every action taken by the student is listed in the Feedback panel (F).

The student can attempt to submit the solution by clicking on “Submit Answer” button (shown at the bottom right). The student’s solution is not considered to be complete until the student has moved all the lines of code from the Problem panel (P) to either the Solution panel (S) or Trash panel (T). Until the student’s solution is complete, whenever the student clicks on “Submit Answer” button, the feedback displayed in the Feedback panel (F) simply asks the student to complete the solution first. Once the solution is complete, it is checked for correctness when the student clicks on the “Submit Answer” button:
• If the solution is correct, a summary of the actions taken by the student is listed in the Feedback panel (F). “Next Problem” button (shown at the bottom right) becomes enabled and the student can click on it to advance to the next puzzle.
• If the solution is incorrect, the first line in the Solution panel (S) that is not in its correct location is highlighted. Feedback is presented in the Feedback panel (F) asking the student to move the line to its correct location later in the code. If the first incorrect line in the Solution panel (S) is a distracter, the feedback also lists why the line of code is incorrect (e.g., “The operator between principal and rate should be multiplication, not division.”) So, the student is given the opportunity to fix the solution one line at a time.

The student can take as many actions as necessary to correct the solution. Once the solution is correct, the student is provided a summary of the actions in the Feedback panel (F) and “Next Problem” button is enabled as described earlier.

If the solution remains incorrect, once the student has executed twice as many actions as the number of lines in the code, “Quit” button becomes enabled (shown at the bottom right), and the student can abandon solving the puzzle by clicking on it. If the student does so, the correct solution to the puzzle is displayed in the Feedback panel (F) and the problem is marked as not having been completed by the student.

So, the only way a student can advance to the next puzzle is by solving it completely and correctly (after zero or more corrections) or by quitting the puzzle.

Help menu is always available – if the student clicks on it, drag-and-drop instructions for solving the puzzle are displayed in a separate window.

The tool allows students to indent/outdent lines of code to improve its readability. But, true to the nature of C++/Java/C# programs, indentation is ignored during grading. If the tool were to be extended to cover Python, the grading scheme would have to be extended to also consider indentation, as is done in 2-D Parsons puzzles (e.g., [8]).

2.2 Unique design: One line at a time

In all prior Parsons puzzle solvers, the student reassembled the code one fragment at a time, a fragment consisting of one or more lines of code. E.g., the following is a fragment in Pascal [13]:

```pascal
writeln( “You win” );
end;
```

Another fragment in the same puzzle is [13]:

```pascal
writeln( “You lose” );
end;
```

Combining multiple lines into a fragment provides two benefits: It eliminates interchangeability when the same line appears multiple times in a program (e.g., end statement in the above two fragments, else clause, open and close braces, etc.): each instance of the repeated line can be incorporated into a unique, non-interchangeable fragment, as shown above. It also addresses the issue that the order of some of the lines in a program is often partial rather than total, e.g., variables could be declared in any order without affecting the semantics of the program. In such cases, lines can be incorporated into fragments so that the order among the fragments is total rather than partial, e.g., all the variable declarations could be encapsulated into one program fragment. So, using fragments considerably simplifies grading in Parsons puzzle solvers.

In our design, the student re-assembles the code one line at a time. This approach has two benefits: it makes the puzzles harder to solve since the number of lines in a program is greater than or equal to the number of fragments. More importantly, since students write code one line at a time, solving a puzzle one line at a time makes the puzzle more epistemic [2]. It forces students to think about the solution in terms of lines which are canonical units of code rather than code fragments which are often contrived aggregations of contiguous lines of code.

Combining multiple lines into a fragment provides two benefits: It eliminates interchangeability when the same line appears multiple times in a program (e.g., end statement in the above two fragments, else clause, open and close braces, etc.): each instance of the repeated line can be incorporated into a unique, non-interchangeable fragment, as shown above. It also addresses the issue that the order of some of the lines in a program is often partial rather than total, e.g., variables could be declared in any order without affecting the semantics of the program. In such cases, lines can be incorporated into fragments so that the order among the fragments is total rather than partial, e.g., all the variable declarations could be encapsulated into one program fragment. So, using fragments considerably simplifies grading in Parsons puzzle solvers.

In our design, the student re-assembles the code one line at a time. This approach has two benefits: it makes the puzzles harder to solve since the number of lines in a program is greater than or equal to the number of fragments. More importantly, since students write code one line at a time, solving a puzzle one line at a time makes the puzzle more epistemic [2]. It forces students to think about the solution in terms of lines which are canonical units of code rather than code fragments which are often contrived aggregations of contiguous lines of code.

Combining multiple lines into a fragment provides two benefits: It eliminates interchangeability when the same line appears multiple times in a program (e.g., end statement in the above two fragments, else clause, open and close braces, etc.): each instance of the repeated line can be incorporated into a unique, non-interchangeable fragment, as shown above. It also addresses the issue that the order of some of the lines in a program is often partial rather than total, e.g., variables could be declared in any order without affecting the semantics of the program. In such cases, lines can be incorporated into fragments so that the order among the fragments is total rather than partial, e.g., all the variable declarations could be encapsulated into one program fragment. So, using fragments considerably simplifies grading in Parsons puzzle solvers.

In our design, the student re-assembles the code one line at a time. This approach has two benefits: it makes the puzzles harder to solve since the number of lines in a program is greater than or equal to the number of fragments. More importantly, since students write code one line at a time, solving a puzzle one line at a time makes the puzzle more epistemic [2]. It forces students to think about the solution in terms of lines which are canonical units of code rather than code fragments which are often contrived aggregations of contiguous lines of code.
tool awards correct credit for placing any instance of the line in any of its correct locations in the code.

We introduced comments in the code to enforce strict order among the lines of code that could otherwise be listed in any order, e.g., in a program to calculate interest on a loan, the loan amount, duration of the loan and interest rate may be input in any order. We introduced comments into the code that could not be reordered, and that enforced a strict order among these three inputs as follows:

// Input the loan amount
...
// Input the rate of interest
...
// Input the duration of loan

This approach of using comments reinforces the good programming practice that code should follow comments. It requires students to associate lines of code with the purpose for which they were written, as expressed in the comments. Finally, comments serve as sub-goal labels, and research shows that sub-goal labels help students solve puzzles statistically better [12].

2.3 Tool Delivery

In prior approaches, the Parsons puzzle solver was built to administer and grade puzzles. We built our tool to also be adaptive, i.e., cater to the puzzle-solving needs of the student.

In our tool, each puzzle is associated with a learning objective. The student’s current knowledge is modeled in terms of these learning objectives. The tool first administers a puzzle on each learning objective, to determine whether the student can solve it completely, correctly and with the fewest actions. Thereafter, for each learning objective on which the student could either not solve the puzzle correctly or took redundant actions to solve it correctly, the tool presents additional puzzles until the student demonstrates mastery, i.e., solves puzzles with no more than a pre-determined number of actions [9].

The adaptive nature of our tool ensures that students do not solve unnecessary puzzles; they solve additional puzzles on only the learning objectives they have not yet met; and their progress in solving puzzles on each learning objective can be documented. Adaptation personalizes the practice provided by the tool, and leverages the benefits of using technology for learning.

2.4 Generation of puzzles

Our tool generates puzzles as randomized instances of parameterized puzzle templates. A puzzle template is written in BNF notation. For example, the template for a variable declaration and initialization statement is:

<T1>T1><V1> = <R1>;

<T1> is a BNF non-terminal that represents a data type, <V1> represents the name of a variable and <R1> represents a random value of data type <T1>. The template may also contain constraints, as in:

<T1>real#> <V1>rate,percent,interest #> = <R1=#<R1=#<12>;

In this template, the data type is constrained to be a real type (e.g., float or double). The possible names for the variable are rate, percent and interest. The value of R1 is constrained to be within the range 3 through 12. So, the line of code generated from this template could be any of the following:

float percent = 5.0;
double rate = 9.0;
float interest = 7.0;

The template of a puzzle is the template of an entire program in the correct order. When generating a puzzle from its template, the tool first replaces all the BNF non-terminals in the template with randomly chosen terminals that obey applicable constraints and are appropriate for the programming language being used (C++, Java or C#). The result is a complete and correct program. The tool next scrambles the lines of code in the program. Finally, the tool adds distractors to the scrambled code.

Distractors are syntactically / semantically buggy versions of the correct code e.g., 5 = count; instead of count = 5;

We used a library of bug specifications to implement distracters. A bug specification contains a regular expression pattern which is used to match each line of code; a code transformation that is used to generate the distracter; a statement that is presented as feedback if the student fails to delete the distracter; and the programming languages and topics to which the bug applies.

For example, consider the following bug specification:

Pattern: ’(.’)+=.(.’)$
Transformation: $1+ =$2
Feedback: Space is not allowed within assignment operator +=
Languages: C++, Java, C#

The line of code count += 5; matches the pattern. As a result, the distracter count + = 5; is generated. If the student fails to discard the distracter, "Space is not allowed within assignment operator +=" is presented as feedback.

The tool contains 100 bug specifications. It generates at most one distracter per line in the code. It can be configured to specify the total number of distracters per puzzle. It can also be configured to place the distracter either at a random location in the code or right before/after the line for which it was generated – the latter placement is known to reduce cognitive load [3].

Initially, the tool generates all possible distracters for each line in the program. Next, it randomly picks a line and randomly picks a distracter for the line. If it is configured to generate n distracters per puzzle, it repeats this process n times, while also making sure that no two distracters are exactly the same.

Since our tool generates puzzles as randomized instances of parameterized templates and adds distracters that are randomly selected from among the available distracters, no two puzzles will be exactly the same even when they have been generated from the same puzzle template. This ensures that no two students will see the same puzzle, and no student will see the same puzzle twice. In addition, each of the 6 modules of the tool contains 15-20 puzzle templates. Therefore, students can practice solving puzzles with the tool as long as and often as they wish.

The tool can be configured to present puzzles in one of two modes:

- Shell-unscrambled mode: The shell of the program is provided to the student in its correct order in the Solution panel (S) at the start of the puzzle-solving session. The student cannot reorder any of its lines. The student is expected to drag and drop only the lines of code contained within the program. Figure 1 shows an example of this mode at the start of a puzzle-solving session.
• None-unscrambled mode: No lines of code are provided unscrambled – the student must reassemble all the lines of code including the shell of the program.

In both the modes, comments in the program are always provided unscrambled in the Solution panel (S) and cannot be reordered. The benefit of shell-unscrambled mode is that students can focus on reconstructing the program and ignore the minimum syntactic baggage needed to make the program compilable, such as import statements in Java. This is the mode in which we have been using the tool.

2.5 Grading of student solution

As mentioned earlier, wherever the order of lines of code is not strict, comments are used to disambiguate their order and impose a strict order. So, each puzzle has only one correct solution. The student is required to solve the puzzle completely and correctly before moving on to the next puzzle. The student can execute any number of actions to solve a puzzle. The number of actions is used as the measure of the student’s ability to solve the puzzle.

In order to solve a puzzle containing n lines of code, a student needs n and no more actions: one action per line to drag and drop the lines of code into their correct location in the Solution panel (S) and one action per line to drag and drop distracters into the Trash panel (T). In order to allow for unintentional drag-and-drop mistakes, the tool was configured to allow a leeway of 10% more actions without penalty. So, anyone who solved the puzzle within 1.1 * n actions was given full credit by the tool. Thereafter, partial credit was awarded inversely proportional to the number of unnecessary actions taken by the student.

Recall that once the student has cleared the Problem panel (P), whenever the student attempts to submit the answer, feedback is provided describing which line is not in its correct location and how it can be fixed. However, students cannot game the system by repeatedly trying to submit an incorrect solution and fixing the solution using the feedback provided to them, because actions over 1.1 * n are negatively graded. This is also why once the student has executed 2*n actions, the tool determines that the student is thrashing and enables Quit button that the student can click to abandon the solution.

3 EXPERIENCE REPORT

The tool is deployed as a Java Web application. It was used by multiple schools each semester in 2015-2017 in their introductory programming course. Students typically used the tool for after-class assignments. They could use each module as often as necessary.

In order to determine whether the ability of the students to solve Parsons puzzles improved with practice, we analyzed the data collected by the module on selection statements over two years. This module presented puzzles on two learning objectives:

1. Programs that used a single if-else statement
2. Programs that used nested if-else statements.

None of the puzzles administered by the module on selection statements used any control construct other than if-else statements. On each learning objective, the module was configured to present 6 different puzzles in a row, as needed by the student: the first puzzle to determine whether the student needed additional practice and subsequent ones to help the student practice solving puzzles. If students got better at solving Parsons puzzles with practice, we expected to see that with each successive (and different) puzzle, we would find a reduction in:

1. the time spent solving the puzzle;
2. the number of actions taken to solve the puzzle.

The number of lines of code varied with the puzzles – some puzzles had more lines of code than others. In order to facilitate comparison of these metrics across different puzzles, we analyzed the time spent per line of the puzzle and number of actions taken per line of the puzzle instead.

The module on selection was used by the students of 24 instructors over the two years data was collected. For our analysis, we used data from the students who had given permission for their data to be used for research purposes. We eliminated those who had solved only one puzzle on a learning objective. For the students who had used the module multiple times, we considered data from only their first complete session.

3.1 Puzzles with a single selection statement

The 6 puzzles administered by selection module on this learning objective, in order, were on the following problems:

1. Read two numbers and print the smaller value among them
2. Read a number and print whether it is odd or even
3. Read sound in decibels and print whether it is loud or not
4. Read body temperature in Celsius, print whether the person has fever
5. Read the price and earnings of a stock and recommend buying/selling the stock based on price to earnings ratio
6. Read two numbers and print the absolute difference between them

Although these puzzles were very different both syntactically and semantically, especially in the eyes of a novice programmer, they all shared the same underlying algorithm.

Table 1 lists the mean of the time in seconds spent by the students per line of code on successive puzzles. The last column lists the statistical significance at 95% confidence level of conducting paired t-test on the times spent on problems n-1 and n for the values of n from 2 to 6. Each successive puzzle was solved by fewer students because of the adaptive nature of the tool. The mean time spent per line of code dropped significantly on each successive puzzle from the first puzzle to the fourth. Subsequent changes in the mean times were not statistically significant.

Table 1: Time spent and actions executed per line of code of each puzzle with a single selection statement

<table>
<thead>
<tr>
<th>Puzzle</th>
<th>Time spent (sec)</th>
<th>Actions executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>p</td>
</tr>
<tr>
<td>1</td>
<td>21803</td>
<td>1.695</td>
</tr>
<tr>
<td>2</td>
<td>16130</td>
<td>1.242</td>
</tr>
<tr>
<td>3</td>
<td>11642</td>
<td>1.159</td>
</tr>
<tr>
<td>4</td>
<td>8294</td>
<td>1.088</td>
</tr>
<tr>
<td>5</td>
<td>10743</td>
<td>1.108</td>
</tr>
<tr>
<td>6</td>
<td>9133</td>
<td>1.163</td>
</tr>
</tbody>
</table>

Table 1 also lists the mean number of actions carried out per line of code on successive puzzles. On the first puzzle, students executed an average of 69% more actions than needed to solve the puzzle. This
mean dropped significantly from the first puzzle through the fourth. Subsequent changes were not statistically significant.

So, on this learning objective, students improved significantly with practice, both in terms of the time spent and the actions executed per line of code through the first four puzzles. There are a few plausible explanations for why this trend did not continue to the fifth and sixth puzzles:

- The sample size N is too small, especially on the sixth puzzle;
- Students who continued to solve puzzles past the fourth puzzle without demonstrating mastery were either less prepared, not engaged or did not stand to benefit from solving puzzles;
- Puzzles on this learning objective take about 8-10 seconds to solve per line of code and about 8-16% more actions than necessary, and no significant further improvements in either metric should be expected.

We plan to test these hypotheses with additional data in the future.

### 3.2 Puzzles with nested selection statements

The 6 puzzles administered by selection module on this learning objective, in order, were on the following problems:

1. Read numerical grade, print the corresponding letter grade A/B/C/D/F (5)
2. Read the month, print the corresponding season (4)
3. Read angle in degrees, print the corresponding quadrant (4)
4. Read altitude in miles, print the name of the corresponding atmospheric layer (5)
5. Read the wavelength of light, print the corresponding color (6)
6. Read wind speed, print the corresponding enhanced Fujita hurricane scale (6)

Listed in parentheses against each problem above is the number of options handled by the code in the puzzle. In each puzzle, the problem statement clarified the ranges needed to correctly solve the puzzle, e.g., red is 620-750 nm, and 158-206 mph is F3 hurricane. Once again, these puzzles were very different both syntactically and semantically, but, they all shared the same algorithm.

Table 2 lists the mean of the time in seconds spent by the students per line of code on successive puzzles. Although the mean time appeared to drop through the fourth puzzle, only the drop from second to third puzzle was marginally significant. The other drops were not statistically significant.

Table 2: Time spent and actions executed per line of code of each puzzle with nested selection statements

<table>
<thead>
<tr>
<th>Puzzle</th>
<th>N</th>
<th>Time spent (sec)</th>
<th>Actions executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>176</td>
<td>18.152</td>
<td>1.561</td>
</tr>
<tr>
<td>2</td>
<td>176</td>
<td>19.404</td>
<td>0.427</td>
</tr>
<tr>
<td>3</td>
<td>145</td>
<td>9.636</td>
<td>0.075</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>8.903</td>
<td>0.132</td>
</tr>
<tr>
<td>5</td>
<td>57</td>
<td>11.663</td>
<td>0.173</td>
</tr>
<tr>
<td>6</td>
<td>23</td>
<td>9.941</td>
<td>0.286</td>
</tr>
</tbody>
</table>

Table 2 also lists the mean number of actions carried out per line of code on successive puzzles. The mean dropped significantly from the first puzzle through the third. Subsequent changes in the mean were not statistically significant except the last one.

The puzzles on this learning objective were conceptually harder than those on the previous learning objective. We still found a significant drop from 56% to 20% redundant actions per line of code from the first to the third puzzle. It could very well be that students need at least 10 seconds per line and at least 20% more actions than necessary to solve puzzles on this learning objective. More data is needed to test this hypothesis.

### 4 DISCUSSION

One of the items of feedback provided by the students who used the tool in the first year was that they would have liked to get a better sense of progress as they went about solving each puzzle. So, we updated the tool to list the number of lines in the puzzle (e.g., 36 in Figure 1) and the number of actions with which it could be solved (e.g., 39 in Figure 1). In addition, the tool enumerates the actions taken by the student so that the student knows how many more actions are available before negative grading kicks in. However, these changes did not seem to positively benefit students [10]. In the future, we also plan to display the edit distance of the student’s solution from the correct solution on a visible scorecard to further motivate students during the puzzle-solving process – when the edit distance reduces to zero, the solution is complete and correct.

The current scheme of grading rewards students who have a clear idea of the purpose of each line of code. It penalizes exploration, wherein a student might try several alternative locations for each line before placing it for the final time. In the future, we plan to implement an alternative grading scheme which will penalize only the actions needed to correct the solution after the student submits a complete solution for the first time. In this scheme, all the exploratory actions taken by the student until the first submission are forgiven. Until then, the only feedback provided by the tool is to direct the student to complete the solution by clearing all the lines of code from the Problem Panel. Currently, the leeway of extra actions allowed by the tool for full credit is 10%. In the future, we plan to investigate whether it is beneficial to adaptively set the leeway based on the collected data (e.g., 15% for the first learning objective in selection and 20% for the second one) and physical needs of the learner.

Our tool helped students solve Parsons puzzles in less time and with fewer missteps. Whether this results in better ability to write code is an interesting hypothesis for future exploration. Another interesting line of enquiry is to mine the collected data to extract patterns of how students solve the puzzles, as has been attempted by other researchers [6]. Finally, we plan to extend the tool to additional topics including functions, arrays and classes. We also plan to extend it to Python.
ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation under grants DUE-1502564 and DUE-1432190.

REFERENCES


Figure 1 © Amruth N. Kumar: Snapshot of Parsons Puzzle Tutor on Selection in Java – Problem statement in Instruction panel (I); Scrambled un-indented code in Problem panel (P); Student solution in Solution panel (S); Deleted lines in Trash panel (T); Feedback in Feedback panel (F). A problem on conversion from numerical grade to letter grade is in progress.