Mnemonic Variable Names in Parsons Puzzles

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

ABSTRACT

In Parsons Puzzles, students are asked to arrange the lines of a program in their correct order. We investigated the effect of using mnemonic variable names in the program on the ease with which students solved the puzzles – whether students were able to solve puzzles containing mnemonic variable names with fewer actions or in less time than single-character variable names. We conducted a controlled study with cross-over design over four semesters. Much to our surprise, we found no statistically significant difference between students solving puzzles with mnemonic variable names versus single-character variable names – either in terms of the number of actions taken, the grade earned or the time spent per puzzle. In this paper, we will describe the experimental setup and data analysis and present the results of the study. We will discuss some hypotheses as to why the readability of the variable names did not impact students' ability to solve Parsons puzzles.

KEYWORDS

Parsons puzzles; Mnemonic variable names; Controlled study

ACM Reference format:
Amruth N. Kumar. 2019. Mnemonic Variable Names in Parsons Puzzles. In *Proceedings of ACM Global Computing Education conference (CompEd'19), May 17–19, 2019, Chengdu, Sichuan, China. ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3300115.3309509

1 Introduction

Parsons puzzles have gained a lot of popularity since their introduction [17]. In a Parsons puzzle, the student is presented a program for a problem, but the lines in the program are scrambled. The student must reassemble the lines in their correct order. The puzzles were designed to be an engaging way to learn programming.

CompEd '19, May 17–19, 2019, Chengdu,Sichuan, China

ACM ISBN 978-1-4503-6259-7/19/05...$15.00
https://doi.org/10.1145/3300115.3309509

Parsons puzzles have since been proposed for use in exams [4], since they are easier to grade than code-writing exercises. At the same time, scores on Parsons puzzles have been found to correlate with scores on code-writing exercises [4]. Researchers have found solving Parsons puzzles to be part of a hierarchy of programming skills alongside code-tracing [15]. 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 [6]. 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 [7]. Software to administer Parsons puzzles have been developed for Turbo Pascal [17], Python (e.g., [3,10]) and C++/Java/C# [12].

Lately, there has been interest in finding patterns in how students go about solving the puzzles [9,11]. Researchers have also looked into what helps students solve the puzzles better, e.g., sub-goal labels help students solve puzzles significantly better [16]; adaptive practice of Parsons puzzles is more efficient while being just as effective as writing code [5]; but motivational supports did not seem to help students while solving puzzles [13]. In this vein, we investigated whether the use of mnemonic variable names in the code had any effect on solving Parsons puzzles. We report the results of our study and discuss their implications.

2 The Study

2.1 Hypothesis

Our research hypothesis was that students would find Parsons puzzles easier to solve when the code in the puzzles used mnemonic variable names rather than single-character variables names (e.g., i, j, k).

Several researchers have documented the importance of mnemonic variable names in programs (e.g., [2,18]): mnemonic variable names improve the readability of a program whereas non-mnemonic single-character variable names make a program harder to read. Mnemonic variable names may lead to better comprehension than single-character variable names [14]. So, we expected that students would be able to solve Parsons puzzles with mnemonic variables faster and with fewer missteps than puzzles with single-character variables. We presented two versions of the same program: one with mnemonic variable names to experimental group and the other, with non-mnemonic single-character variable names to control group, while keeping

all the other factors such as indentation, commenting and structure the same between the two versions.

2.2 Tools

For this study, we used epplets (epplets.org), a Parsons puzzle tool [12]. The tool presents the scrambled lines of code in the left panel, called Problem panel, and has the student reassemble the lines of code in their correct order in the right panel, called Solution Panel using drag-and-drop action. Students can also delete a line of code by dragging it into Trash panel (Please see [12] for a figure of the user interface). The student is required to solve a puzzle completely and correctly before going on to the next puzzle. The tool provides feedback to help the student fix an incorrect solution. The tool also allows the student to bail out of solving a puzzle when hopelessly lost.

The tool requires that the student reassemble the code one line at a time, instead of one program fragment at a time [17]. So, given a program with n lines of code, a student can solve the puzzle with n drag-and-drop actions. The tool presents comments in the code *in situ* in the solution panel. Students are expected to drag and drop lines of code under appropriate comments.

For this study, we had the students solve Parsons puzzles on two topics: `while` loops and `for` loops.

2.2.1 `while` Loop Puzzle

We had students solve two puzzles on `while` loops. The first puzzle was used to get students accustomed to the user interface of the tool. So, all the students were presented code with mnemonic variable names on the first puzzle. The puzzle presented code for the problem: "Read numbers till the same number appears back to back. Print the first number to appear back to back (e.g., 4 appears back to back in 3,7,5,7,4,4,5 and is printed)."

The second puzzle presented on `while` loops was used to conduct this study. It was for the problem: "Input the face of a card. Next, read a deck of cards and print how many cards into the deck the input card is found, followed by its successor. For example, if the input card is 6, in a deck that starts with the cards 1,8,6,10,7,9,13, the 6 card is in 3rd place and 7 card is in 5th place."

The single-character variable code provided to control group on the second puzzle was as follows (in C++):

```
#include <iostream>
using namespace std;
int main()
{
  // Declare y
  long y;
  // Declare r
  long r;
  // Declare b
  long b = 1;
  // Read into y the face of the card to look for in the deck
  cout << "Enter the face of the card to look for in the deck (1-13)";
  cin >> y;
  // Find the card and its successor in a deck of cards
  cout << "Enter the cards in the deck one at a time";
  cin >> r;
  while( r != y )
  {
    cin >> r;
    b = b + 1;
  } // End of while loop from line 24
  cout << "Card " << y
       << " found in deck at position " << b;
  y = y + 1;
  cin >> r;
  b = b + 1;
  while( r != y )
  {
    cin >> r;
    b = b + 1;
  } // End of while loop from line 33
  cout << "Card " << y
       << " found in deck at position " << b;
} // End of function main
```

Note that the code used single-character variable names, with the characters having no mnemonic association with the purposes they served. The corresponding mnemonic variable code presented to experimental group was as follows (in C++):

```
#include <iostream>
using namespace std;
int main()
{
  // Declare selectCard
  int selectCard;
  // Declare cardDeck
  int cardDeck;
  // Declare counter
  int counter = 1;
  // Read into selectCard the face of the card to look for in the deck
  cout << "Enter the face of the card to look for in the deck (1-13)";
  cin >> selectCard;
  // Find the card and its successor in a deck of cards
  cout << "Enter the cards in the deck one at a time";
  cin >> cardDeck;
  while( cardDeck != selectCard )
  {
    cin >> cardDeck;
    counter = counter + 1;
  } // End of while loop from line 24
  cout << "Card " << selectCard
       << " found in deck at position " << counter;
  selectCard = selectCard + 1;
  cin >> cardDeck;
  counter = counter + 1;
  while( cardDeck != selectCard )
  {
```

```
    cin >> cardDeck;
    counter = counter + 1;
  } // End of while loop from line 33
  cout << "Card " << selectCard
       << " found in deck at position " << counter;
} // End of function main
```

In the two versions of the code presented before, the longest stretch of code re-assembled by students without the benefit of any comments is highlighted in bold.

2.2.2 `for` Loop Puzzle

Once again., we had students solve two puzzles on `for` loops. The first puzzle was used to get students accustomed to the user interface of the tool. So, all the students were presented mnemonic variable code on the first puzzle. The puzzle presented code for the problem: "Read two numbers. Calculate the sum of all the numbers between the two and print it, e.g., if 4 and 7 are read, print 22, which is the sum of 4,5,6 and 7."

The second puzzle presented on `for` loops was used to conduct this study. It was for the problem: "Read the monthly income for a year. Print its sum. Read the monthly expenses for the year. Print money left over after expenses."

The single-character variable code provided to control group on the second puzzle was as follows (in C++):

```
#include <iostream>
using namespace std;
int main()
{
  // Declare x
  long double x;
  // Declare r
  long double r;
  // Declare a
  unsigned long a;
  // Read monthly income into x, print sum in r
  r = 0;
  for( a = 1; a <= 12; a ++ )
  {
    cout << "Please enter the income for month " << a;
    cin >> x;
    r = r + x;
  } // End of for loop from line 19
  cout << "Sum of monthly income is $ " << r;
  // Read monthly expenses into x, print balance in r after expenses
  for( a = 1; a <= 12; a ++ )
  {
    cout << "Please enter the expenses for month " << a;
    cin >> x;
    r = r - x;
  } // End of for loop from line 30
  cout << "Balance after expenses is $ " << r;
} // End of function main
```

The corresponding mnemonic variable code presented to experimental group was as follows, wherein, the longest stretch of code re-assembled by students without the benefit of comments is highlighted in bold:

```
using namespace std;
int main()
{
  // Declare amount
  float amount;
  // Declare balance
  float balance;
  // Declare counter
  unsigned short counter;
  // Read monthly income into amount, print sum in balance
  balance = 0;
  for( counter = 1; counter <= 12; counter ++ )
  {
    cout << "Please enter the income for month "
         << counter;
    cin >> amount;
    balance = balance + amount;
  } // End of for loop from line 19
  cout << "Sum of monthly income is $ " << balance;
  // Read monthly expenses into amount, print balance in balance after expenses
  for( counter = 1; counter <= 12; counter ++ )
  {
    cout << "Please enter the expenses for month " << counter;
    cin >> amount;
    balance = balance - amount;
  } // End of for loop from line 30
  cout << "Balance after expenses is $ " << balance;
} // End of function main
```

2.3 Protocol

We conducted a crossover study. We divided students into two groups: A and Z. Their treatments on `while` and `for` loop puzzles were as shown in Table 1.

Table 1: Treatment for Groups A and Z on `while` loop and `for` loop Parsons puzzles

Group	`while` loop	`for` loop
A	Single-Character	Mnemonic
Z	Mnemonic	Single-Character

The puzzles used in this study were the second puzzles students solved on `while` and `for` loops. This ensured that students would have overcome any user interface issues by the time they solved these puzzles.

2.4 Variables

Students were required to completely and correctly solve each puzzle. The independent variable in the study was the variable naming scheme in the puzzle presented to the student: mnemonic versus single-character.

We used four dependent variables:

- The number of steps taken by the student to solve the puzzle. The steps included moving a line of code from the problem panel to the solution panel, reordering a line within the solution panel, and deleting a line from the problem or solution panel to the trash panel.
- The grade on the puzzle, calculated as 100% if the student solved the puzzle with as many steps as the number of lines in the code. If the student took more steps than the number of lines in the code, each superfluous step was penalized against one correct step, e.g., if the program contained 20 lines and the student took 30 steps to solve the puzzle completely and correctly, the student got credit for 10 steps out of 20. So, the normalized score awarded to the student was 10 / 20 = 0.5. The normalized score was bound to the range 0 → 1.0. This negative grading scheme meant that a student could score 0 on a puzzle even after having solved it correctly.
- The time taken by the student to solve the puzzle completely and correctly, in seconds.
- The time taken per step by the student to solve the puzzle completely and correctly, calculated as time / number of steps taken.

2.5 Data Collection

We collected data over four semesters: Fall 2016 – Spring 2018. The subjects were students in the introductory programming course, both majors and non-majors. The puzzles were provided as two of a dozen after-class assignments. The number of students who solved Parsons puzzles on `while` and `for` loops in each treatment over the four semesters is listed in Table 2. Group A consisted of students from 5 baccalaureate institutions, 2 community colleges and 2 high schools and Group Z from 6 baccalaureate institutions and one community college.

Students had the option to solve the puzzles on the two topics as many times as they wanted. For our analysis, we considered data from only the first time a student solved puzzles on either topic. Since this was a crossover study, we considered only the students who had served as both control and experimental subjects, i.e., we eliminated students who had not solved all four puzzles: two each on `while` and `for` loops. For the same reason, we also eliminated students who had bailed out of solving any of the four puzzles. After these eliminations, group A consisted of 34 students and Group Z consisted of 40 students.

Table 2: Number of students who solved Parsons puzzles in each condition over four semesters

	Single-Character	Mnemonic
while	67	82
for	75	65

2.6 Data Analysis

On each topic (`while` and `for` loop), we compared the control and experimental group performance on the first puzzle to check if the two groups were comparable – both the groups were presented the same code with mnemonic variables for the first puzzle. We used data from the second puzzle to compare mnemonic versus single-character variable treatments: control group was presented single-character variable code and experimental group was presented mnemonic code.

Table 3 lists the mean and 95% confidence interval of the number of steps taken, the normalized score, the time in seconds, and the time taken per step by the two treatment subjects on the first `while` loop puzzle. ANOVA analysis yielded no significant difference between the two groups for steps [F91,73) = 0.6, p = 0.44], grade [F(1,71) = 1.05, p = 0.31], time [F(1,73) = 1.84, p = 0.18] or time per step [F(1,73) = 1.66, p = 0.20]. *So, the two treatment groups, when provided the same treatment, were comparable.*

Table 3: Comparison of the two groups on the first `while` loop puzzle with the same treatment

`while` loop	Single-Character (Group A, N = 34)	Mnemonic (Group Z, N=40)
Steps	24.65 ± 3.68	22.70 ± 3.40
Grade	0.62 ± 0.13	0.71 ± 0.12
Time	352.92 ± 65.24	292.55 ± 60.15
Time/Step	14.27 ± 1.88	12.61 ± 1.73

Table 4 lists the same figures for the two groups on the second puzzle on `while` loops, wherein, control group (A) was presented single-character code and experimental group (Z) was presented mnemonic variable code. ANOVA analysis yielded no significant difference between the two groups for steps [F(1,73) = 0.03, p = 0.86], grade [F(1,73) = 0.02, p = 0.88], time [F(1,73) = 0.10, p = 0.75] or time per step [F(1,73) = 0.17, p = 0.68]. In other words, *the use of mnemonic variable names had no impact on the number of steps taken to solve the puzzle, the score earned on the puzzle or the time taken to solve the puzzle.*

Table 4: Comparison of the two groups on the second `while` loop puzzle with differential treatments.

`while` loop	Single-Character (Group A, N = 34)	Mnemonic (Group Z, N = 40)
Steps	46.29 ± 7.65	47.23 ± 7.05
Grade	0.42 ± 0.13	0.41 ± 0.12
Time	690.38 ± 103.1	667.75 ± 95.05
Time/Step	15.76 ± 2.28	15.12 ± 2.10

Table 5 lists the performance of control (Z) and experimental (A) groups on the first `for` loop puzzle, where both were provided the same treatment, viz., mnemonic variable code. Once again, ANOVA analysis yielded no statistically significant difference between the two treatments on the steps [F(1,73) = 0.11, p = 0.74], grade [F(1,73) = 2.18, p = 0.14], time [F(1,73) = 0.28, p = 0.60] or time taken per step [F(1,73) = 0.67, p = 0.42]. So, once again, the

two groups were comparable in their performance when provided the same treatment, viz., mnemonic variable code.

Table 5: Comparison of the two groups on the first `for` loop puzzle with the same treatment

`for` loop	Single-Character (Group Z, N= 40)	Mnemonic (Group A, N = 34)
Steps	22.98 ± 1.90	22.50 ± 2.06
Grade	0.82 ± 0.07	0.89 ± 0.08
Time	237.25 ± 42.57	253.79 ± 46.17
Time/Step	10.38 ± 1.95	11.57 ± 2.12

Finally, Table 6 lists the figures for the second `for` loop puzzle, wherein, control group (Z) was presented single-character variable code and experimental group (A) was provided mnemonic variable code. ANOVA analysis yielded no statistically significant difference between the treatments on steps [$F(1,73) < 0.01$, $p = 0.94$], grade [$F(1,73) < 0.01$, $p = 0.93$], time [$F(1,73) = 1.01$, $p + 0.32$] or time taken per step [$F(1,73) = 1.28$, $p = 0.26$]. So, mnemonic variables in *the code had no impact on the performance of the students.*

Table 6: Comparison of the two groups on the second `for` loop puzzle with differential treatments

`for` loop	Single-Character (Group Z, N= 40)	Mnemonic (Group A, N = 34)
Steps	37.20 ± 3.46	37.38 ± 3.76
Grade	0.59 ± 0.11	0.58 ± 0.12
Time	443.0 ± 56.0	484.68 ± 60.73
Time/Step	12.17 ± 1.37	13.32 ± 1.49

Next, we conducted ANCOVA analysis of the grade on the second puzzle, with treatment as the fixed factor and grade on the first problem as a covariate. For this analysis, we combined the data from both the loops. We found no main effect for treatment [$F(1,145) = 0.51$, $p = 0.48$]: the grade with single-character variable version was 0.53 ± 0.08 compared to 0.49 ± 0.08 with mnemonic variable version. Similarly, ANCOVA analysis of the time taken per step on the second puzzle with the time taken per step on the first puzzle as covariate yielded no significant effect for treatment [$F(1,147) = 0.28$, $p = 0.60$]: subjects spent 13.82 ± 1.27 seconds per step with single-character variable version compared to 14.30 ± 1.27 seconds with mnemonic variable version. *So, even after accounting for variations in student performance on the first puzzle, we found no effect of treatment on their performance on the second puzzle.*

We compared the performance on the first puzzle with that on the second puzzle on each topic. On each topic, everyone was presented mnemonic variable version on the first puzzle, but only one of the two groups (A/Z) was presented mnemonic version on the second puzzle while the other group was presented single-character version. Since the puzzles were different, and involved different numbers of lines of code, we compared only the time taken per step. If mnemonic variables decreased puzzle-solving time, the group that was presented single-character version on the second puzzle would have spent significantly more time on the second puzzle compared to the first puzzle than the mnemonic variable group. The time spent per step for the two groups on the two puzzles in the two topics are listed in Table 7. Repeated measures ANOVA analysis yielded no significant interaction between the puzzle and treatment on `while` loops [$F(1,72) = 1.07$, $p = 0.30$] or `for` loops [$F(1,72) = 1.31$, $p = 0.26$]. *So, working with mnemonic variable code on one puzzle did not influence the performance of students on a subsequent puzzle.*

Table 7: Repeated measures comparison of the time taken per step on the first and second puzzles for the two groups

`while` loop	Single-Character (Group A, N = 34)	Mnemonic (Group Z, N=40)
Puzzle 1	14.27 ± 1.88	12.61 ± 1.73
Puzzle 2	15.76 ± 2.28	15.12 ± 2.10
`for` loop	Single-Character (Group Z, N= 40)	Mnemonic (Group A, N = 34)
Puzzle 1	10.38 ± 1.95	11.57 ± 2.12
Puzzle 2	12.17 ± 1.37	13.32 ± 1.49

We computed the average grade on the first puzzle on each topic and used it to group students into two: less-prepared students who scored below average and better-prepared students who scored average or above. One-way ANOVA of the grade on the second puzzle with treatment and preparedness as fixed factors yielded interaction between the two factors as shown in Table 8, but it was not statistically significant. Similar analysis of the time taken per step yielded a significant interaction between the two factors [$F(1,147) = 7.475$, $p = 0.007$]: less-prepared students spent more time with mnemonic treatment than single-character treatment whereas better-prepared students spent more time with single-character treatment than mnemonic treatment. This was not an artifact of the puzzle topic (`while` versus `for`) because the interaction with topic was not significant. So *mnemonic variables in Parsons puzzle code may differentially affect students based on their level of preparation.*

Table 8: The effect of treatment on the grade and time taken per step on the second puzzle by less- versus better-prepared students

Grade	Single-Character	Mnemonic
Less	0.40 ± 0.13 (33)	0.46 ± 0.16 (22)
Better	0.60 ± 0.11 (41)	0.50 ± 0.10 (52)
Time per step	Single-Character	Mnemonic
Less	11.81 ± 1.94 (33)	15.49 ± 2.38 (22)
Better	15.44 ± 1.74 (41)	13.79 ± 1.55 (52)

2 Results and Discussion

We expected that students would be able to solve Parsons puzzles faster and with fewer steps when the puzzles contained mnemonic variables instead of single-character variables. But, the results of the study did not support this hypothesis, much to our surprise. We considered various explanations for this outcome.

The puzzles used in the study involved 3 variables each. It could be argued that the readability of a program is not impaired if it contains only three variables that are poorly named. But, poorly named variables make it harder to track the flow of data in the program, especially in the section of the code that involves back-to-back loops – the section boldfaced in the listings presented earlier. This is true even for experienced programmers, not just novices. Nevertheless, we plan to repeat the study with puzzle programs involving many more than 3 variables.

It could be argued that the programs presented in the puzzles are short: 19-24 lines long. But, they are complicated enough for beginning programmers. It could be argued that the comments provided by the tool in the solution panel make it easy to assemble some of the lines of code such as variable declarations and input statements. But, some comments in the program are followed by 9 – 19 lines of back-to-back code, wherein, students had to reassemble code without any assistance from comments. Since these uncommented lines carry out the actual computations in the program, assembling them is the hardest part of each puzzle.

A third possible explanation is that mnemonic variables indeed facilitate solving Parsons puzzles, but the effect size is so small that we need much larger sample sizes to evaluate the hypothesis - power analysis yielded observed power of 5% to 14% for many of the analyses. We plan to repeat the controlled experiment every semester, and plan to revisit this study with additional data in the future.

In a recent study, professional developers were presented two versions of six library code segments: one with mnemonic variables and the other with meaningless single-character variable names. In three of the code segments, no statistically significant difference was observed between the two treatments in terms of code comprehension [1]. Authors of the study attributed this surprising result to the use of poorly chosen mnemonic names. It turns out, choosing mnemonic names is not as objective an exercise as one might like – the probability that two people choose the same name for a variable was found to be less than 20% in one study [8]. Even if universally acceptable mnemonic names are used, they may hinder program comprehension by serving as misleading "beacons" (code elements that illuminate the code's function) when novices hold an incorrect model of the purpose of the program [1].

The counter-arguments to these issues that confound the utility of mnemonic names are: 1) we chose mnemonic names from the problem statement provided for each puzzle (e.g., `cardDeck`) or listed them in the comments preceding each section (e.g., `amount`); and 2) unlike in the earlier study [1] wherein, programmers had to guess the purpose of each code segment, we described the purpose of each Parsons puzzle program in the accompanying problem statement. So, even though we found mnemonic variable names did not seem to provide any benefits and this result concurs with some of the results found in the previous study [1], the reasons why are not the same. Given the counter-intuitive nature of our result, this study should be reproduced in different settings before any definitive conclusions are drawn.

We expect students to build a mental model of the semantics of the program as they solve a Parsons puzzle: tracing each variable through its lifecycle, tracing the flow of data and control through the program, and thereby, understanding how each line of code fits into the overall program. If so, mnemonic variables would make it easier to build such a model by making it easier to trace each variable through its lifecycle and trace the flow of data through the program. Single-character variables on the other hand force the reader to scan the program repeatedly to re-establish the purpose of each unhelpfully named variable.

May be students resort to techniques other than constructing a complete mental model of the program to solve Parsons puzzles. So, the use of mnemonic variables neither helps nor hurts their ability to solve the puzzles. If a student can solve a Parsons puzzle without building a mental model of the underlying program first, it is essential for researchers to isolate and identify the alternative puzzle-solving techniques, and see whether and how those techniques contribute to the learning of programming. In the future, we plan to test whether students re-assemble the lines of code in a puzzle in random order, or in an order influenced by the semantics of the lines of code.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Eran Avidan and Dror G. Feitelson. 2017. Effects of variable names on comprehension an empirical study. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 55-65. DOI: https://doi.org/10.1109/ICPC.2017.27.

[2] Scott Blinman and Andy Cockburn. 2005. Program comprehension: investigating the effects of naming style and documentation. In Proceedings of the Sixth Australasian conference on User interface - Volume 40 (AUIC '05), Mark Billinghurst and Andy Cockburn (Eds.), Vol. 40. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 73-78.

[3] Nick Cheng and Brian Harrington. 2017. The Code Mangler: Evaluating Coding Ability Without Writing any Code. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 123-128. DOI: https://doi.org/10.1145/3017680.3017704.

[4] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 113-124. DOI=http://dx.doi.org/10.1145/1404520.1404532.

[5] Barbara J. Ericson, James D. Foley, and Jochen Rick. 2018. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. ACM, New York, NY, USA, 60-68. DOI: https://doi.org/10.1145/3230977.3231000

[6] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. 2015. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In *Proceedings of the eleventh annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 169-178. DOI: https://doi.org/10.1145/2787622.2787731.

[7] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. 2017. Solving Parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17)*. ACM, New York, NY, USA, 20-29. DOI: https://doi.org/10.1145/3141880.3141895.

[8] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. 1987. The vocabulary problem in human-system communication. Communications of the.ACM 30,(November 1987), 964-971 DOI= https://doi.org/10.1145/32206.32212

[9] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How do students solve parsons programming problems?: an analysis of interaction traces. In *Proceedings of the ninth annual international conference on International computing education research (ICER '12)*. ACM, New York, NY, USA, 119-126. DOI: https://doi.org/10.1145/2361276.2361300.

[10] Petri Ihantola and Ville Karavirta. 2010. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education (ITiCSE '10)*. ACM, New York, NY, USA, 302-302. DOI: https://doi.org/10.1145/1822090.1822178

[11] Petri Ihantola and Ville Karavirta. 2011.Two-Dimensional Parson's Puzzles: The Conceot, Tools, and First Observations. *Journal of Information Technology Education: Innovations in Practice*. Vol 10. 2011. 119-132. DOI= https://doi.org/10.28945/1394

[12] Amruth N. Kumar. 2018. Epplets: A Tool for Solving Parsons Puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 527-532. DOI: https://doi.org/10.1145/3159450.3159576.

[13] Amruth N. Kumar. 2017. The Effect of Providing Motivational Support in Parsons Puzzle Tutors. I*n Proceedings of Artificial Intelligence in Education. (AI-ED 2017)*, Wuhan, China, June 2017, 528-531. DOI= https://doi.org/10.1007/978-3-319-61425-0_56

[14] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06). IEEE Computer Society, Washington, DC, USA, 3-12. DOI: https://doi.org/10.1109/ICPC.2006.51

[15] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 101-112. DOI=http://dx.doi.org/10.1145/1404520.1404531.

[16] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals Help Students Solve Parsons Problems. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 42-47. DOI: https://doi.org/10.1145/2839509.2844617.

[17] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06), Denise Tolhurst and Samuel Mann (Eds.), Vol. 52. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157-163.

[18] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with identifiers and comments in source code comprehension and maintenance: results from an ethnographically-informed study with students and professionals. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14). ACM, New York, NY, USA, , Article 48 , 10 pages. DOI: http://dx.doi.org/10.1145/2601248.2601251