# Representing and Evaluating Strategies for Solving Parsons Puzzles

Amruth N. Kumar [0000-0002-1951-3995]

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

**Abstract.** Parsons puzzles are popular for programming education. Identifying the strategies used by students solving Parsons puzzles is of interest because they can be used to determine to what extent students use the strategies typically associated with programming expertise, and to provide feedback and monitor the progress of students in a tutor. We propose solution sequence as an approximation of the student's strategy for solving Parsons puzzles. It is scalable in terms of both the size of the puzzle and the number of students solving the puzzle. We propose BNF grammar to represent desirable puzzle-solving strategies associated with expert programmers. This representation is extensible and agnostic to the puzzle-solving strategies themselves. Finally, we propose a best match parser that matches a student's solution sequence against the BNF grammar of a desirable strategy and quantifies the degree to which the student's solution conforms to the desirable strategy. As a proof of concept, we used the parser to analyze the data collected by a Parsons puzzle tutor on if-else statements over five semesters and found a significant difference between C++ and Java puzzle-solvers in terms of their conformance to one desirable puzzle-solving strategy. Being able to tease out the effects of individual components of a strategy is one benefit of our approach: we found that relaxing shell-first constraint in the strategy resulted in significant improvement in the conformance of both C++ and Java students.

**Keywords:** Parsons puzzle, Puzzle-solving strategy, Context Free Grammar, Evaluation.

## 1 Introduction

Parsons puzzles were first proposed to "provide students with the opportunity for rote learning of syntactic constructs" in Turbo Pascal [13]. In a Parsons puzzle [13], the student is presented a problem statement, and the program written for it. The lines in the program are provided in scrambled order. The student is asked to re-assemble the lines in their correct order.

Parsons puzzles have been proposed for use in exams [2], since they are easier to grade than code-writing exercises, and yet, scores on Parsons puzzles correlate with scores on code-writing exercises [2]. 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 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 [3]. So, Parsons puzzles have been gaining popularity for use in introductory programming courses.

Each Parsons puzzle has only one correct solution. So, the correct solution, i.e., the final re-assembled program will be the same for all the students. However, the temporal order in which students go about assembling the lines of code will vary among the students. This order indicates their solution strategy influenced by their understanding of the syntactic and semantic relationships among the lines of code, e.g., assembling a declaration statement before an assignment statement; or assembling the entire shell of a control statement before filling in its contents.

Recently, there has been interest among researchers in identifying the solution strategies used by students when solving Parsons puzzles. One study on Python Parsons puzzles [7] observed that some students re-assembled the lines using "linear" order, i.e., the random order in which the lines were provided. These researchers also observed backtracking and looping behavior, which were unproductive. Another preliminary study on Python Parsons puzzles observed that experts used top-down strategy to solve the puzzles [8], i.e., function header first, followed by control statements and eventually, individual statements. In another study of a Python Parsons puzzle tutor [5], researchers found that one common strategy was to focus on types of program statements. Novices often grouped lines based on indentation, whereas experts often built the solution top-down, demonstrating a better understanding of the program model. This study used think-aloud protocol and audio/video recordings to identify puzzle-solving strategies of novices and experts.

The benefits of identifying puzzle-solving strategies of students are manifold. Research shows that the strategies used by novices are different from those used by experts for programming tasks [15]. Experts use a strategy of reading a program in the order in which it is executed, and this leads to the development of a hierarchical mental model [12]. Expert programmers show more evidence of using a hierarchical mental model when understanding a well-written program than novices [6]. Moreover, a novice's success in learning to program is influenced by the student's mental model of programming [1,14]. So, identifying a student's puzzle-solving strategy helps:

- determine whether the student is using the strategies typically used by experts;
- provide feedback to nudge the student towards adopting the successful strategies used by experts with the expectation that doing so will help the student develop the mental models associated with programming expertise, and thereby become a better programmer himself/herself, which is the goal of using Parsons puzzles; and
- determine whether the puzzle-solving strategy of students improves with practice when they use a tutor.

In order to identify the solution strategies used by students when solving Parsons puzzles, we propose 1) **solution sequence** as an approximate linear representation of their puzzle-solving behavior; 2) **BNF grammar** as a flexible representation of desirable solution strategies; 3) a **best-match parser** that matches a student's solution sequence against the BNF grammar for the puzzle to quantify the student's conformance to a desirable solution strategy. As a proof of concept, we apply our approach to analyze the solution strategies of students in data collected by a Parsons puzzle tutor on `if-else` statements.

## 2 The Solution

### 2.1 Action Sequence, Inner Product and Solution Sequence

Think-aloud protocol has been used to identify puzzle-solving strategies [5,8]. But, this protocol is expensive and not scalable to larger numbers of students. The other approach used to identify strategies is the conversion of interaction traces (log data) into state diagrams [7, 8], where states represent snapshots of the solution in progress. But, this approach leads to combinatorially explosive number of states for any puzzle that contains more than a few lines of code, and is hence, not scalable to larger puzzles.

Instead, we propose to represent the student's puzzle-solving behavior as the temporal order in which the lines of code in the solution are assembled in their correct spatial location, e.g., if the puzzle contains 5 lines, and the student starts by placing line 3 in its correct location, followed by lines 1, 5, 2 and 4 in their correct locations, we represent the puzzle-solving behavior of the student as the **solution sequence** [3, 1, 5, 2, 4].

Solution sequence is itself derived from the **action sequence** of the student, which is the sequence of actions carried out by the student to solve a Parsons puzzle. We make a simplifying assumption to generate the solution sequence from action sequence. A student may move a line of code multiple times in the process of solving the puzzle. But, in order to generate the student's solution sequence, we consider only the last time the student moves the line before arriving at the correct solution for the puzzle, e.g., suppose the student moves the lines of a puzzle containing 5 lines in the following order: 3, 5, 4, 2, 1, 5, 2, 4, which is the student's action sequence. The corresponding solution sequence is calculated as the **inner product** of the action sequence and the time of submission of the correct solution (shown in Figure 1). The resulting solution sequence is [3, 1, 5, 2, 4] corresponding to the order in which each of the 5 lines was last placed in its correct location in the solution. This inner product transformation eliminates from the solution sequence backtracking and looping behavior found in action sequence – so, it loses information about unproductive behaviors of the student [7]. But, it retains information about the sequence of decisions the student takes about the final placement of those lines just as the correct solution falls into place. In other words, it captures the thinking of the student in putting the solution together after any trial-and-error activities such as backtracking and looping. It is a reflection of the syntactic and semantic relationships the student sees among the lines of code in the puzzle, i.e., it is a reflection of the student's mental model of the program. Therefore, it is an approximation of the puzzle-solving strategy of the student.

For a puzzle containing n lines of code, the length of action sequence is greater than or equal to n. The length of solution sequence is n. If a student solves a puzzle with no redundant actions, the solution sequence of the student is the same as the action sequence.

So, instead of considering combinatorially explosive number of states, we consider a solution sequence of linear complexity, whose size is the number of lines in the code. This solution sequence can be automatically extracted from the log data collected by the tutor as compared to manually eliciting it using think-aloud protocol. So, our approach scales both with the number of lines in the puzzle and the number of students solving the puzzle.

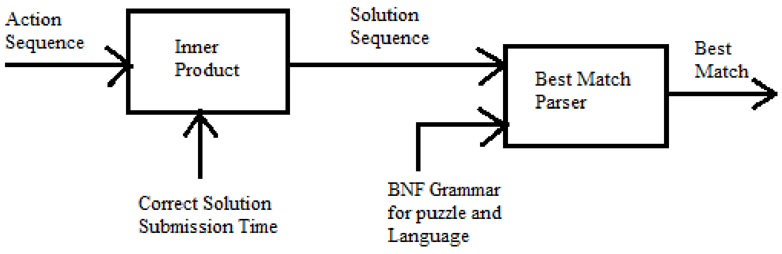

**Fig. 1.** Architecture of our approach

## 2.2 Desirable Solution Strategies

Experts have been found to use top-down strategy to solve Parsons puzzles [5,8]. When reading a program, experts do so in the order in which it is executed, which leads to the development of a hierarchical mental model [12], a model that conceptualizes the elements of the program as forming a layered network of components, each of which can be of arbitrary depth and breadth [10]. Expert programmers also use a hierarchical mental model as compared to novices when understanding a program [6]. So, some desirable puzzle-solving strategies are those that use a top-down strategy and/or a hierarchical mental model of the program.

It is not clear that there is only one ideal strategy for solving Parsons puzzles – even experts have been seen to switch strategies [5]. Therefore, any approach for incorporating puzzle-solving strategies associated with programming expertise into Parsons puzzle tutors must accommodate a variety of strategies.

A generalized mathematical model of puzzle-solving strategies is characterized as follows: The lines in a puzzle may be grouped into subsets, such that all the lines within a subset must be assembled contiguously in time. The order among the subsets as well as among the lines within each subset may be total, partial or none. For example, two of the subsets within a puzzle may be: the set of declaration statements and the set of lines containing the shell of an `if-else` statement. A good strategy for solving the puzzle might require that declaration statements must be assembled before `if-else` statement (total order). The various lines of code within the set of declaration statements may be assembled in any order (no order). The braces within the shell of an `if-else` statement must be assembled in partial order: the closing brace must be assembled after the opening brace, but the braces of if-clause and else-clause may be assembled in either order.

Given these properties, we propose to use context free grammar or Backus-Naur Form (BNF) grammar to represent desirable puzzle-solving strategies. Once desirable strategies are represented using BNF grammar, a parser can be used to determine the degree to which the solution sequence of a student conforms to one or more desirable strategies. Since the BNF grammar can be used to simultaneously represent any number of strategies for a puzzle, whether they are complementary or contradictory, using it does not require commitment to any single idealized solution strategy.

## 2.3 BNF Grammar

A BNF grammar contains rules. Each rule is made up of terminals and non-terminals. Terminals are elements of the language, e.g., eat, drink. Non-terminals are types of the elements, e.g., noun, verb. Each rule contains a left hand side and a right hand side. The left hand side of a rule is a single non-terminal. The right hand side is a sequence of terminals and non-terminals. Alternatives in a rule are separated by vertical stroke.

In our case, the elements of the language are line numbers. The non-terminals are types of lines in the puzzle, e.g., declaration, output. Each rule specifies the temporal order in which a type of lines is assembled. Consider the following `if-else` statement to print two numbers in ascending order in C++:

```
1     if( first < second )
2     {
3        cout << first << endl;
4        cout << second << endl;
5     }
6     else
7     {
8        cout << second << endl;
9        cout << first << endl;
10    }
```

In the code, lines 1,2,5,6,7 and 10 constitute the shell of the `if-else` statement. Lines 3 and 4 are if-clause. Lines 8 and 9 are else-clause. A good programming practice is to assemble the entire shell of the `if-else` before assembling the if-clause and else-clause. This practice may be expressed in BNF grammar using the following rules:

```
<if-else> → <shell> <if-clause> <else-clause> | <shell> <else-clause> <if-clause>
<shell> → 1 2 5 6 7 10 | 1 6 2 5 7 10 | 1 6 7 10 2 5
<if-clause> → 3 4 | 4 3
<else-clause> → 8 9 | 9 8
```

In the grammar, non-terminals are enclosed in angle brackets <>. The first rule states that when assembling the `if-else` statement, the recommended practice is to assemble the entire shell first, followed by if-clause and else-clause in either order. Similarly, the third rule states that when assembling if-clause, lines 3 and 4 can be placed in their correct location in either order. According to the grammar, every possible combination of rules represents a desirable solution strategy. So, the above grammar represents 2 X 3 X 2 X 2 = 24 different solution strategies.

The grammar is extensible. If it is later determined that another good strategy would be to assemble the braces enclosing if-clause and else-clause at the same time as the clauses, adding the following rules to the grammar will accommodate the new strategy:

```
<if-else> → 1 <if-block> 6 <else-block> | 1 6 <if-block> <else-block>
<if-block> → 2 <if-clause> 5
<else-block> → 7 <else-clause> 10
```

Each BNF grammar is specific to a puzzle and programming language (See Figure 1). So, if a tutor contains 10 puzzles and can be used for 3 different programming languages, 30 different BNF grammars must be encoded to analyze the data collected by the tutor.

### 2.4 A Best Match Parser

A parser takes a sentence and a grammar as inputs and outputs whether the sentence is correct according to the grammar. In our case, the sentence is a solution sequence. We not only wanted to know whether a solution sequence conformed to a grammar, but also the degree to which it conformed if it did not fully conform to the grammar. So, we developed a parser that takes two inputs: a BNF grammar and a solution sequence. It returns a number representing the **best match**, i.e., the maximum number of consecutive lines in the solution sequence that conform to any combination of rules in the grammar for the puzzle (See Figure 1). It uses depth-first search to do so.

For example, given the grammar in Section 2.3, the behavior of the parser is as follows:

- The solution sequence [1, 2, 5, 6, 7, 10, 8, 9, 3, 4] is completely correct. So, the parser returns 10, the length of the solution sequence.
- The solution sequence [1, 6, 2, 5, 7, 10, 3, **9**, 4, 8] is correct up to line 9 - the student did not complete assembling if-clause before moving on to else-clause. The parser returns 7, the number of lines correct up to line 9.
- The solution sequence [1, 6, 7, 10, **5**, 2, 3, 4, 8, 9] is correct up to line 5 – the student assembled the closing brace before the opening brace enclosing if-clause in the `if-else` shell. The parser returns 4, the number of lines correct up to line 5.

Given a puzzle of n lines, the best match returned by the parser is in the range [0 … n].

## 3 A Proof-of-Concept Evaluation

As proof of concept, we evaluated the data collected by a Parsons puzzle tutor [9] (epplets.org) on `if-else` statements over 5 semesters: Fall 2016 – Fall 2018. The tutor was used by students in introductory programming courses as after-class assignment.

Students used the tutor to solve puzzles in C++ or Java. Data for analysis was included from students who gave IRB permission for their data to be used in the study.

The first puzzle solved by all the students was on a program to read two numbers, and print the smaller value among them. Java version of the program is shown below. The program contains 14 manipulable lines in both C++ and Java: 2 lines of variable declaration (lines 9 and 11 below), 2 lines per input for 2 inputs (lines 13, 14 and 16, 17), followed by 8 lines of `if-else` statement (lines 19-26). The other lines in the program, such as comments, were provided *in-situ.*

```
 1   // The Java program - 2005
 2   import java.util.Scanner;
 3   public class Problem
 4   {
 5      public static void main( String args[] )
 6      {
 7         Scanner stdin = new Scanner( System.in );
 8         // Declare firstNum
 9         int firstNum;
10         // Declare secondValue
11         int secondValue;
12         // Read firstNum
13         System.out.print( "Enter the first value");
14         firstNum = stdin.nextInt();
15         // Read secondValue
16         System.out.print( "Enter the second value");
17         secondValue = stdin.nextInt();
18         // Print the smaller value
19         if( firstNum  <  secondValue )
20         {
21            System.out.print( firstNum);
22         } // End of if-clause
23         else
24         {
25            System.out.print( secondValue);
26         } // End of else-clause
27      }  // End of method main
28   } // End of class Problem
```

The BNF grammar for the puzzle stipulated that it should be solved in the following order: the two declaration statements in any order, the two inputs in any order, `if-else` shell, followed finally by if-clause and else-clause in any order. The BNF grammar for Java is listed below.

```
<start-2005>  → <declaration> <input> <output>
<declaration> → 9 11 | 11 9
<input>       → <input1> <input2> | <input2> <input1>
<input1>      → 13 14 | 14 13
<input2>      → 16 17 | 17 16
<output>      → <if-frame> <clauses>
```

```
<if-frame>    → 19 <if-brace> 23 <else-brace> | 19 <if-brace> 23 |
                19 23 <braces> | 19 23
<braces>      → <if-brace> <else-brace> | <else-brace> <if-brace> |
                <if-brace> | <else-brace>
<if-brace>    → 20 22
<else-brace>  → 24 26
<clauses>     → <if-clause> <else-clause> | <else-clause> <if-clause>
<if-clause>   → 21
<else-clause> → 25
```

We conducted one-way ANOVA with the best match as the dependent variable and programming language as the fixed factor. We found a significant main effect for programming language [F(1,411) = 15.794, p < 0.001]: C++ students had a best match score (5.62 ± 0.58, N = 114) significantly greater than Java students (4.24 ± 0.36, N = 298). The mean best match for C++ students corresponded to assembling declaration statements and both the inputs in the correct order. The same for Java students corresponded to assembling declaration statements and only the first input in the correct order. One possible explanation is that Java students were more likely to have been exposed to graphical user input, where inputs are separated spatially, So, they did not appreciate the linear order imposed on inputs in console input.

Our approach can be used to tease out the benefits of individual components of desirable strategies. We hypothesized that students might not appreciate the benefit of assembling the entire shell of an `if-else` statement before assembling the code contained in if-clause and else-clause. So, we added the following rules to the grammar that bypassed this restriction and allowed students to assemble the braces around if-clause and else-clause while assembling those clauses.

```
<output>      → 19 <if-block> 23 <else-block> | 19 23 <if-block> <else-block>
<if-block>    → 20 21 22 | 20 22 21 | 21 20 22
<else-block>  → 24 25 26 | 24 26 25 | 25 24 26
```

We re-analyzed the data to calculate the best match with this extended grammar. ANOVA analysis with the original and extended best matches as the repeated measure and programming language as the fixed factor yielded a significant within-subjects effect for the change in grammar [F(1,410) = 197.367, p < 0.0001]: the mean best match increased from 4.932 ± 0.34 to 6.863 ± 0.56. As could be expected from earlier results, we found a significant between-subjects effect for language also [F(1,410) = 18.599, p < 0.001]. The best match of C++ students (N=114) significantly improved from 5.623 ± 0.58 to 8.123 ± 0.955. Similarly, the best match of Java students (N=298) significantly improved from 4.242 ± 0.36 to 5.604 ± 0.59. So, bypassing the requirement that the shell of `if-else` must be assembled completely before its contents resulted in a mean improvement in conformance of 2.5 lines for C++ students and 1.4 lines for Java students. *So, if assembling the shell first is indeed a good strategy, there is a need to better educate students about its importance.*

## 4 Discussion

We proposed **solution sequence** as an approximation of the student's strategy for solving Parsons puzzles. It is scalable in terms of both the size of the puzzle and the number of students solving the puzzle, compared to earlier approaches for modeling puzzle-solving strategies of students [5,7,8].

We proposed **BNF grammar** to represent desirable puzzle-solving strategies associated with expert programmers. This representation can accommodate complementary and contradictory strategies together, and is extensible. This representation scheme is agnostic to the puzzle-solving strategies themselves.

Finally, we proposed a best match parser that matches a student's solution sequence against the BNF grammar of a desirable strategy and quantifies the degree to which the student's solution conforms to the desirable strategy. As a proof of concept, we used the parser to analyze the data collected by a Parsons puzzle tutor on `if-else` statements over five semesters and found a significant difference between C++ and Java puzzle-solvers in terms of their conformance to one desirable puzzle-solving strategy. We demonstrated how our approach can be used to tease out the benefits of individual components of a desirable strategy. In the process, we also found that bypassing the constraint that shell must be assembled before its contents resulted in significant improvement in conformance of both C++ and Java students. Our approach can be used for any programming language or paradigm.

One shortcoming of our approach is that solution sequence is an approximation of the puzzle-solving strategy used by students because it loses information such as looping and backtracking behavior [7]. So, while it is suitable for understanding puzzle-solving strategies and mental models used by students, action sequence is better for diagnosing student misconceptions.

Our approach is designed to model desirable strategies and quantify how much students' solutions conform to those strategies, not to find patterns in students' solutions. Algorithms such as frequency counts and k-means clustering are better suited for finding patterns in students' solutions.

Currently, the best match parser returns the maximum number of consecutive lines in a solution sequence that match a desirable strategy. In the future, we will consider alternative matching algorithms such as Levenshtein algorithm.

In the future, we plan to use the BNF grammar representation of desirable puzzle-solving strategies, coupled with a generator (instead of a parser) to provide proactive hints to students as they solve puzzles in a tutor. Such hints might help students learn the puzzle-solving strategies associated with programming expertise and in turn, help students develop the mental models used by expert programmers and become better programmers themselves.

## Acknowledgments

Partial support for this work was provided by the National Science Foundation under grants DUE-1432190 and DUE-1502564. The author thanks Anthony Bucci for suggesting the use of BNF grammars and Alessio Gaspar, Paul Wiegand and Jennifer Albert for associated discussions.

## References

1. Cañas, J.J., Bajo, M.T. & Gonzalvo, P.: Mental models and computer programming. International Journal of Human-Computer Studies, vol. 40 (5), pp. 795-811 (1994)
2. Denny, P., Luxton-Reilly, A., and Simon, B.: Evaluating a new exam question: Parsons problems. In Proc. Fourth international Workshop on Computing Education Research (ICER '08), pp. 113-124. ACM, New York, NY, USA (2008)
3. Ericson, B.J., Margulieux, L.E., and Rick, J.: Solving Parsons problems versus fixing and writing code. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17), pp. 20-29. ACM, New York, NY, USA (2017)
4. Ericson, B.J., Guzdial, M.J., and Morrison, B.B.: Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In Proc.11th annual International Conference on International Computing Education Research (ICER '15), pp. 169-178. ACM, New York, NY, USA (2015)
5. Fabic, G., Mitrovic, A., Neshatian, K.: Towards a mobile Python tutor: understanding differences in strategies used by novices and experts. In: Proceedings of the 13th International Conference on Intelligent Tutoring Systems, LNCS, vol. 9684, pp. 447–448. Springer Heidelberg (2016)
6. Fix, V., Wiedenbeck, S. and Scholtz, J.: Mental representations of programs by novices and experts. In Proc. INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93), pp. 74-79. ACM, New York, NY, USA (1993)
7. Helminen, J., Ihantola, P., Karavirta, V. and Malmi, L.: How do students solve parsons programming problems? An analysis of interaction traces. In Proc. Ninth annual international conference on International computing education research (ICER '12), pp. 119-126. ACM, New York, NY, USA (2012)
8. Ihantola, P. and Karavirta, V.: Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations. Journal of Information Technology Education: Innovations in Practice, vol. 10, pp. 1–14. (2011)
9. Kumar, A.N.: Epplets: A Tool for Solving Parsons Puzzles. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), pp. 527-532. ACM, New York, NY, USA (2018)
10. Letovsky, S.: Cognitive processes in program comprehension. In Soloway, E. and Iyengar, S. (eds.) Empirical Studies of Programmers, pp 58-79. Ablex, Norwood, NJ (1986)
11. Lopez, N., Whalley, J., Robbins, P. and Lister, R.: Relationships between reading, tracing and writing skills in introductory programming. In Proc. 4th international Workshop on Computing Education Research (ICER '08), pp. 101-112. ACM, New York, NY, USA (2008)
12. Nanja, M. and Cook, C. R.: An analysis of the online debugging process. In Olson, G.M., Sheppard, S. and Soloway, E. (eds.) Empirical Studies of Programmers: Second Workshop, pp. 172-184. Ablex, Norwood, NJ (1987)
13. Parsons, D and Haden, P.: Parson's programming puzzles: a fun and effective learning tool for first programming courses. In Proc. 8th Australasian Conference on Computing Education (ACE '06), Vol. 52. pp 157-163. Australian Computer Society, Inc. (2006)
14. Soloway, E. & Ehrlich, K.: Empirical studies of programmer knowledge. IEEE Transactions of Software Engineering, vol. SE-10 (5), pp. 595-609. (1984)
15. Winslow, L. E.: Programming pedagogy—a psychological overview. ACM SIGCSE Bulletin, vol. 28(3), pp. 17-22. (1996)