

Understanding the Effects of Using Parsons Problems to Scaffold Code Writing for Students with Varying CS Self-Efficacy Levels

Xinying Hou
University of Michigan
Ann Arbor, Michigan, USA
xyhou@umich.edu

Barbara J. Ericson
University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

Xu Wang
University of Michigan
Ann Arbor, Michigan, USA
xwanghci@umich.edu

ABSTRACT

Introductory programming courses aim to teach students to write code independently. However, transitioning from studying worked examples to generating their own code is often difficult and frustrating for students, especially those with lower CS self-efficacy in general. Therefore, we investigated the impact of using Parsons problems as a code-writing scaffold for students with varying levels of CS self-efficacy. Parsons problems are programming tasks where students arrange mixed-up code blocks in the correct order. We conducted a between-subjects study with undergraduate students (N=89) on a topic where students have limited code-writing expertise. Students were randomly assigned to one of two conditions. Students in one condition practiced writing code without any scaffolding, while students in the other condition were provided with scaffolding in the form of an equivalent Parsons problem. We found that, for students with low CS self-efficacy levels, those who received scaffolding achieved significantly higher practice performance and in-practice problem-solving efficiency compared to those without any scaffolding. Furthermore, when given Parsons problems as scaffolding during practice, students with lower CS self-efficacy were more likely to solve them. In addition, students with higher pre-practice knowledge on the topic were more likely to effectively use the Parsons scaffolding. This study provides evidence for the benefits of using Parsons problems to scaffold students' write-code activities. It also has implications for optimizing the Parsons scaffolding experience for students, including providing personalized and adaptive Parsons problems based on the student's current problem-solving status.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction; Interactive learning environments**; • **Social and professional topics** → **Computing education**.

KEYWORDS

Parsons problems, Scaffolding, Code writing, Undergraduate CS, Hint, Introductory Programming, Self-Efficacy

ACM Reference Format:

Xinying Hou, Barbara J. Ericson, and Xu Wang. 2023. Understanding the Effects of Using Parsons Problems to Scaffold Code Writing for Students with Varying CS Self-Efficacy Levels. In *23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23)*, November 13–18, 2023, Koli, Finland. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3631802.3631832>

1 INTRODUCTION

Commonly used techniques to introduce a new programming topic in college lectures often involve direct instruction and worked example code demonstration [31]. After this, students are expected to gain expertise by solving more programming problems independently. However, while students appear to understand the theoretical concepts and examples taught in lectures, transitioning to writing full solutions to new problems remains a huge challenge [16, 28, 30, 47]. As a result, they often fail to overcome the challenge of writing their own code independently, particularly students with low CS self-efficacy, who are less likely to persevere when faced with difficulties [52, 57].

To tackle this issue, prior work has investigated different scaffolding approaches to assist students in learning to write code. Scaffolding refers to the assistance given to someone to help them complete a task when they cannot do it independently yet [26]. One-on-one tutoring, which can provide the desired scaffolding, has been found to be more effective than traditional classroom instruction with one instructor [5]. However, as computer science becomes increasingly popular, the number of students in undergraduate CS classes has grown considerably in recent years, which in turn presents serious challenges to offering one-on-one tutoring due to the high cost [21].

Parsons problems are an increasingly popular type of programming exercise that requires students to place mixed-up code blocks in the correct order [13, 41]. Previous research has shown that Parsons problems generally require less cognitive load from students compared to write-code problems [22] and facilitate greater engagement [15]. This inspired our work to investigate whether Parsons problems can be used as effective scaffolding when students are writing code for new topics. As a type of completion problem, they have the potential to help students transition from worked examples to conventional write-code programming problems [40, 51].

To understand the effectiveness of Parsons problems as a scaffolding technique when students are learning to write code, we conducted a between-subjects classroom experiment. In the experiment, we wanted to understand whether and how students with distinct levels of self-efficacy differ in practice and learning when they received Parsons problems as scaffolding (Parsons scaffolding

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Koli Calling '23, November 13–18, 2023, Koli, Finland

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1653-9/23/11...\$15.00
<https://doi.org/10.1145/3631802.3631832>

- PS condition) during write-code practice versus not (NP condition). Furthermore, we performed in-depth analyses to understand students' experiences when interacting with Parsons problems as a scaffolding technique, and the relationship between their interaction and CS self-efficacy levels. We examined the following research questions:

- RQ1.1: Are there differences between conditions in terms of practice performance, problem-solving efficiency, and posttest performance for students with low CS self-efficacy levels?
- RQ1.2: Are there differences between conditions in terms of practice performance, problem-solving efficiency, and posttest performance for students with high CS self-efficacy levels?
- RQ2: In the Parsons Problems as Scaffolding (PS) condition, how did students with varying CS self-efficacy levels use the Parsons scaffolding?
- RQ3: In the Parsons problems as Scaffolding (PS) condition, how did students rate the usefulness of the Parsons scaffolding and why?

2 RELATED WORK

2.1 Scaffolding Write-Code Problems

Scaffolding strategies help students finish a task or build new understanding so that they can perform comparable activities on their own later. By providing desired scaffolding, Bloom demonstrated that one-on-one human tutoring helps students improve their learning by two standard deviations over typical classroom instruction with a single teacher for 30 students [5]. However, providing this type of support in high student-to-teacher ratio courses, such as introductory CS courses, can be too expensive.

To address this issue, researchers have looked into computer-assisted scaffolding techniques at various stages of code development. One line of research provides scaffolding to restrict the entry difficulty of code-writing to avoid cognitive overload. For instance, Denny et al. showed that letting students review and think about problem statements before writing any code had a positive influence on performance [10]. Similarly, Garcia tested design-based Parsons problems which asked students to put strategic plans in order, and found that some students used these problems to try to understand the problem better but that others just used a trial and error approach to solve the problem [18].

Another line of research focuses on providing scaffolding throughout the code-writing process, such as by providing next-step hints in Intelligent Tutoring Systems (ITS). For example, Rivers built ITAP, which employs a three-stage process to generate next-step hints for student code submissions [45]. Her initial analysis found that students with hints spent less time practicing but achieved the same learning outcomes. Nevertheless, the inconsistency in the quality of the automated hints is still a problem, affecting students' trust in these systems and future help-seeking behaviors [43]. Furthermore, automated hints rarely contain comprehensive guidance, such as examples, that might be leveraged to overcome the "design barriers" experienced by beginner programmers [27]. As a result, by using low-level hints before starting to program a task, some novices experienced inefficient help-seeking outcomes

[33]. As we aim to provide scaffolding for students in the early stages of acquiring coding skills on this topic, we would like to implement a more comprehensive scaffolding method during the code-writing process, which is providing students with equivalent Parsons problems alongside the write-code problems.

2.2 Existing work on Parsons problems

In the original design of Parsons problems, Parsons and Haden provided students with a problem description and a set of drag-and-drop code fragments [41]. Each code fragment was made up of one or more lines, and some of the lines included incorrect code. To complete a problem, students chose the correct code segments and arranged them in the correct order. They reported that the majority of students thought this type of problem was useful for learning [41].

Subsequent research has produced a range of Parsons problem varieties, and the difficulty levels of these formats vary depending on the tasks students need to complete and the code blocks presented. For instance, in one-dimensional Parsons problems, code blocks must be organized in the right vertical sequence, while in two-dimensional Parsons problems, the blocks must additionally be appropriately indented [25]. In addition, Weinman et al. proposed faded Parsons problems where students must use valid expressions to fill in blanks in the lines of code and rearrange the lines of code to create a proper program [53]. Distractors, code blocks that are not needed in a correct solution, can also be added to make the problems more challenging [14]. Distractor blocks typically include syntactic or semantic flaws. There are two types of display for distractor blocks: paired distractors, which contain some indication that students have to pick one of a set, and unpaired distractors, which are randomly mixed in with the correct blocks. Previous research has demonstrated that paired distractors make Parsons problems easier to solve than unpaired distractors [9].

Ericson et al. created two types of adaptation for Parsons problems: intra-problem and inter-problem adaptation [12]. Intra-problem adaptation reduces the difficulty of the current Parsons problem, and inter-problem adaptation affects the difficulty of the next problem based on students' performance on the current problem [14]. Learners can initiate the intra-problem adaptation by clicking the "Help Me" button after at least three full attempts, and the system will either remove a distractor block or combine two blocks into one. Inter-problem adaptation happens after the system evaluates the learner's performance on the previous Parsons problem. To make the next problem easier, it will remove or pair distractors with the correct blocks. To make it harder, it will add all distractors or randomly mix them with the correct blocks. In this study, we use two-dimensional Parsons problems with intra-problem adaptation and paired distractors.

2.3 Cognitive Load Theory

According to cognitive load theory, the human cognitive architecture is made up of numerous memory stores, including a restricted working memory and an unlimited long-term memory [46]. Working memory is limited in terms of capacity and duration, especially when processing new information. Cognitive load refers to the

working memory resources necessary when learning new information [49]. Two core cognitive load categories are intrinsic and extrinsic cognitive load [50]. Intrinsic cognitive load relates to the material's inherent difficulty, which is mediated by the learner's prior knowledge [50]. Instructional strategies like segmenting and pre-training can be applied to manage intrinsic cognitive load due to overly complex content [36]. Extraneous cognitive load is determined by how the instructional information is delivered and what the learner is expected to perform [50]. The extraneous load can be reduced by dealing with typical instructional elements that may cause extraneous load [7].

Computer programming is a highly cognitive skill that requires mastering multiple competencies and is recognized as being inherently difficult to learn, making cognitive load theory one of the most relevant theories in computing education research [4]. Previous studies have adopted a wide range of instructional recommendations provided by cognitive load theory to programming learning. Among those effects, the use of worked examples is extremely popular in introductory programming courses [4]. Worked examples demonstrate an expert's comprehensive solution to a problem, allowing students to learn how to solve problems before they can write correct code independently. They are commonly used for novice learners to reduce cognitive load [48]. However, in the traditional way of employing programming examples, there are dramatic shifts in cognitive demand when moving directly from fully completed examples to solving a problem by writing code from scratch [44]. To bridge this gap, one recommended method is to use completion problems with a partial answer before expecting students to complete a full problem [50]. Since Parsons problems provide students with the right code blocks but still require them to arrange them in the right order, they fall under the category of completion problems. Therefore, by adding Parsons problems as scaffolding to code writing problems, we expect to implement a smoother transition for students from studying examples to solving write-code problems independently.

2.4 Self-efficacy in CS Learning

Self-efficacy describes people's perceptions of their own abilities to complete a task [3]. Self-efficacy is important in education as individuals' self-efficacy can influence their willingness to put effort into a task, decisions about future involvement in a task, and attitudes when facing obstacles [2]. For example, students with high self-efficacy are usually more enthusiastic about participating in and completing learning tasks than students with low self-efficacy. When facing difficulties, students who believe in their abilities (high self-efficacy) do not avoid difficult tasks but view them as challenges that must be overcome. For example, science learners with a greater degree of self-efficacy will put in more effort on learning activities and will persevere when faced with difficulties, resulting in learning success [6, 57].

When it comes to the CS domain, self-efficacy has become among the most studied constructs to understand programming learning outcomes and persistence in computing learning and careers [42]. Specifically, Wiedenbeck discovered that self-efficacy was positively connected with two distinct programming course outcomes: performance in debugging tasks and the overall course grade [55]. In another study, Lewis et al. interviewed 31 students at two public

universities and discovered that one crucial aspect in their decision to major in CS was their perception of their CS ability (self-efficacy in CS) [29]. Similarly, Miura applied self-efficacy surveys and found students with higher self-efficacy were more likely to enroll in a computer science course in the college [37]. In this work, we will investigate how students with varying levels of CS self-efficacy used the Parsons scaffolding during practice and whether there are differences between conditions in terms of practice performance, in-practice problem-solving efficiency, and posttest performance for students with various CS self-efficacy levels.

2.5 Using Parsons problem to Scaffold Writing Code

Parsons problems have been explored for both formative assessment (practice) and summative assessment [13]. Prior studies have provided evidence that most students find Parsons problems engaging [41], and students can achieve the same level of learning as writing the equivalent code, but with higher learning efficiency [14]. Morrison et al. also reported that Parsons problems were more sensitive than writing code for assessing students' learning gains [38]. Parsons problems allow students to demonstrate their understanding of the meaning and sequence of programs, which helps to assess students' knowledge in ways that writing code alone cannot.

An initial study of Parsons problems as scaffolding explored when, why, and how students apply Parsons problems to scaffold their write-code problems [24]. The think-aloud study found that students opened the Parsons problem at three different stages: planning, implementing, and debugging a solution. In addition, Hou et al. identified four distinct ways in which students successfully interacted with Parsons problems to help them complete coding tasks: "scan Parsons problem", "attempt Parsons problem", "solve and replace their code", and "solve and modify their code." [24]. In "scan Parsons problem", the learner looks at the unsolved Parsons problem for ideas about how to get started or for particular information, but does not attempt to solve it. In "attempt Parsons problem", the learner makes some effort to solve the Parsons problem but does not completely solve it before finishing the write-code problem. In "solve and replace their code", learners solve the Parsons problem and use that solution to replace any code they wrote in the write-code problem. In "solve and modify their code", learners solve the Parsons problem and then modify their solution in the write-code problem. When experimenting with the learning effectiveness of Parsons problems to scaffold write-code problems, Hou et al. discovered that students who were given Parsons problems as scaffolding for code writing problems took less time to complete those problems, however, there was no learning improvement from pretest to posttest in either condition, indicating that those students had already mastered the topic [24]. In contrast to the prior study, we chose a more advanced programming topic to which students had little prior exposure. Our research investigated if Parsons problems can help students bridge the gap between learning from worked examples and solving write-code problems independently.

3 METHOD

Our IRB-approved study was conducted in the fall semester of 2022 at a large public research university in the northern United

States. All participants were enrolled in a data-oriented programming course, which was the second required Python course for the university's information science majors, though other majors take the course as well. This course covered programming concepts including Python basic data structures (list, tuple, and dict), object-oriented programming concepts (classes, objects, and inheritance), how to debug, how to use unit testing, basic web scraping, regular expressions, HTML, XML, JSON, working with APIs, working with databases, and Matplotlib.

3.1 System Interface

Runestone allows students to write, execute code, and receive immediate feedback from unit test results (Figure 1) [17]. Our study added an equivalent optional Parsons problem to each write-code problem to scaffold students' code-writing practice. When students have difficulty solving a write-code problem independently, they could display, interact with, and work on the equivalent Parsons problem in a preview window (Figure 2), but they were still required to solve the write-code problem to earn points. Students could not copy and paste the Parsons solution to the write-code problem, they had to retype it. When students failed to pass all the unit tests after three submission attempts, they would receive a pop-up prompt reminding them that help is available ("Help is Available Using the Toggle Question Selector").

Write a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect', 150)` would print "Respect, 150".

```

1 class Song:
2     def __init__(self, title, len):
3         self.title = title
4         self.len = len
5     def __str__(self):
6         return self.title + ", " + str(self.len)
7
8

```

Result	Actual Value	Expected Value	Notes
Fail	'Respect, 150'	'Respect, 150'	testing __str__ with Respect
Fail	'Truth..., 183'	'Truth..., 183'	testing __str__ with Truth Hurts

You passed: 0.0% of the tests

Figure 1: Screenshot of a write-code problem with the unit test results

3.2 Participants and Procedure

The classroom study was conducted during the 80-minute lecture period in week three of the class; students who did not attend the lecture were allowed to finish the study by the end of the day. A total of 134 students participated in this study. Students were randomly assigned to one of two conditions: Parsons-Scaffolding condition (PS) and No-Parsons-Scaffolding condition (NP). Students in the Parsons-Scaffolding (PS) condition received a text-entry write-code interface with an equivalent two-dimensional adaptive Parsons problem as scaffolding (Figure 2), while the No-Parsons Scaffolding (NP) group only had the write-code interface (Figure 1).

Students first read an introduction to Python classes. It covered three fundamental concepts: defining a new class, constructing new objects, and writing new methods. Each concept included textual instruction, an executable worked example followed by a task where students were instructed to modify the code (Figure 3). After finishing the introduction to the concepts, students received an introduction to the types of problems in the system: scaffolded code writing or non-scaffolded. Following that, students were asked to complete a programming self-efficacy survey (6 questions) and a self-evaluation on their knowledge about writing code for classes (4 questions). Then, students were given four write-code practice problems in each of the two conditions, with the only difference being that students in PS condition had the equivalent Parsons problems as scaffolding. Students in PS condition were explicitly told that they had to enter code in the write-code area to earn points. We generated a random number between 1 to 10 once the student clicked to start the practice. Based on this number, we assigned them to the NP condition if it was odd and the PS condition if it was even. After each practice question, students in the PS condition were given a question, asking them to rate the perceived usefulness of using a Parsons problem to help them solve the write-code problem on a Likert scale from 1-low to 9-high. Students who did not use the Parsons scaffolding were asked to skip the corresponding survey question.

Additionally, following the last practice question, PS students were asked to respond to a brief open-ended question explaining their general perceived usefulness of a Parsons problem as scaffolding while writing code. Students in both conditions then finished the posttest. Each section had no time restrictions, allowing students to progress through the materials at their own pace until the end of the day. The final sample contained 89 students who completed all of the materials in order (41 in PS and 48 in NP). Only three students chose to finish the materials outside of class time. They were not outliers after checking their main measures, including practice performance, problem-solving efficiency, and posttest scores.

Problems

Toggle Question: Parsons Mixed-Up Code - Classes, Basic_Song_pp

Write a class `Song` with an `__init__` method that takes a `title` as a string and `len` as a number and initializes these attributes in the current object. Then define the `__str__` method to return the `title, len`. For example, `print(s)` when `s = Song('Respect', 150)` would print "Respect, 150".

Drop from here

```

1 class Song:
2     def __init__(self, title, len):
3         self.title = title
4         self.len = len
5
6
7
8
9
10
11
12
13
14 s = Song('Respect', 150)
15 print(s)

```

Drop blocks here

```

16 def __init__(title, len):
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

Check Reset Help me

Figure 2: Screenshot of using a Parsons problem to scaffold a write-code problem

3.3 Materials

There were five parts to the materials: basic knowledge instruction on how to create classes, methods, objects in Python; a survey to measure students' CS self-efficacy level and pre-practice knowledge level about writing class; four write-code practice problems

Table 1: Questions about student general CS self-efficacy level

Question Item
1 - Generally I have felt secure about attempting computer programming problems.
2 - I am sure I could do advanced work in computer science.
3 - I am sure that I can learn programming.
4 - I think I could handle more difficult programming problems.
5 - I can get good grades in computer science.
6 - I have a lot of self-confidence when it comes to programming.

Table 2: Questions and ratings on prior-practice knowledge levels in Python class

Question Item	Rate Level
1 - Creating classes like <code>class Person:</code> and objects like <code>p = Person("XXX")</code>	1 - I am unfamiliar with these concepts
2 - Methods like <code>__init__</code> and <code>__str__</code>	2 - I know what they mean, but have not used them in a program
3 - The use of <code>self</code> in <code>class</code>	3 - I have used these concepts in a program, but am not confident about my ability to use them
4 - Defining instance variables like <code>self.color = color</code>	4 - I am confident in my ability to use these concepts in simple programs
	5 - I am confident in my ability to use these concepts in complex programs

and corresponding equivalent Parsons problems; and an immediate posttest to assess their learning performance.

3.3.1 Basic knowledge instruction. The knowledge introduction described how to create a new `class`, how to override the inherited `__init__` and `__str__` methods, how to define a new method, and how to create objects. For each subtopic, we provided a combination of textual introduction and an interactive worked example that the participant could execute and revise to learn how to handle a specific problem in Python [1]. Students in this study had little prior experience with creating a new `class` in Python, so we believed this basic knowledge instruction part would provide students with some conceptual and procedural knowledge on this topic (Figure 3).

3.3.2 Survey Items. To gather students' self-efficacy in computer science learning in general (their general CS self-efficacy level), we used the self-efficacy items developed for the computer science domain [56]. The scale to measure self-efficacy was the first 6 items from a 13-item subscale developed by Wiebe et al. [54]. Participants were asked to rate these statements on a five-point scale, ranging from "1-strongly disagree" to "5-strongly agree" (Table 1). This resulted in a score scale of 6 to 30.

Additionally, to assess whether there was a significant difference between the two groups, we used Duran et al.'s method [11] to collect self-ratings for the pre-practice knowledge level on this topic. Self-evaluation instruments can be used to capture pedagogically useful information about students' prior programming knowledge [11]. To measure students' pre-practice knowledge level, we created a four-question survey around `class` and asked students to rate how well they understood the concepts and could proceed with translating them into actual code (Table 2). For the rate levels, given that this instrument was used to measure a new topic, we created five different levels adapted from the original prototype

self-evaluation instrument (Table 2), and assigned a score of 1-low to 5-high for each level, with a total score range of 4 to 20.

3.3.3 Write-code Practice Problems. We sourced the write-code problems from an intermediate Python programming course at a public research university in the United States. We chose four write-code practice problems, each worth 10 points (Table 3). The first problem required students to create a class with an `__init__` and `__str__` method. The second and third problems involved implementing a third method in addition to `__init__` and `__str__`. The fourth problem was the hardest one which required performing a random selection from a list, in addition to creating a class and a method. Our goal was to make sure that participants with different skill levels all had a chance to use a Parsons problem to scaffold a write-code problem. During practice, students from the two conditions both received execution-based feedback after each run, which showed compiler errors and the output from running the code including the results from unit tests (Figure 1). Each problem was scored out of 10 points based on the percentage of unit tests that passed. There were a total of 40 possible points for the overall practice. In this work, we first extracted the practice results provided by the auto-grading system. Then, we manually checked the final code submissions and accounted for any auto-grading errors before conducting the analyses.


3.3.4 Parsons Problems as Scaffolding. One recent study of Parsons problems found that a Parsons problem with an unusual solution increased students' cognitive load [22]. To address this problem, we clustered student-written code from previous semesters using the OverCode software [19], and then used the most common student solution cluster to create an equivalent Parsons problem (Figure 2). To highlight common misconceptions, we also inserted paired distractors into the Parsons problems. Experts created the distractor blocks based on common syntax or semantic errors [41]. Additionally, a "Help Me" button was provided at the bottom of each Parsons

problem to assist students who struggle while solving the Parsons problem. This button triggered intra-problem adaptation, which either removes a distractor or combines two blocks into one if the learner had made at least three attempts to solve the Parsons problem and there are more than three blocks left in the solution.

3.3.5 Posttest Items. Two types of questions were included in the posttest: write-code problems (10 points each, 20 points in total) and fix-code problems (10 points each, 20 points in total). In a write-code problem, the student needs to write the correct code from scratch following the problem description, and in a fix-code problem, the student must fix the errors in the existing buggy solution. These questions covered similar concepts to the write-code practice questions. Every time they ran the code, students would receive execution-based feedback, which showed compiler errors and the output from running the code, usually including results from unit tests. We calculated their posttest scores by using the proportion of passed unit tests in the final submission.

Look the code below. It defines a class. It also declares methods which are functions that are defined inside of a class. One of the methods, `__init__`, is automatically called when a new object is created by the class. One of the methods, `__str__`, is automatically called when you print an object of the class.

A Book Class



```

1 class Book:
2     """ Represents a book object """
3
4     def __init__(self, title, author):
5         self.title = title
6         self.author = author
7
8     def __str__(self):
9         return "title: " + self.title + " author: " + self.author
10
11 def main():
12     b2 = Book("A Wrinkle in Time", "M. L'Engle")
13     print(b2)
14     b1 = Book("Goodnight Moon", "Margaret Wise Brown")
15     print(b1)
16
17 main()

```

Activity: 1 A class to represent a book (class_tog_book_act1_v2.fl)

Figure 3: Screenshot of the first part of the basic knowledge instruction

4 RESULTS

4.1 RQ1: Are there differences between conditions in terms of practice performance, problem-solving efficiency, and posttest performance for students with low CS self-efficacy levels (RQ1.1) and for students with high CS self-efficacy levels (RQ1.2)?

Descriptive statistics on student responses to the survey items are included in Table 4. As shown in Table 4, on average, students in both conditions reported a pre-practice knowledge level between "I know what it means, but have not used it in a program" (8 total points) and "I have used this concept in a program, but am not confident about my ability to use it" (12 total points). This demonstrated that, after receiving basic direct instruction with worked examples, these students gained some confidence in their ability to write code around `class`. However, they have not become experts on this topic. This indicated that those participants reached the expected skill level for this study. In addition, as the conditions

(NP and PS) were randomly assigned, we expected students in both conditions to have a similar level of self-rated prior knowledge in Python `class` and general CS self-efficacy before they started the practice. Given that their self-rated pre-practice knowledge and general CS self-efficacy were not normally distributed, we applied two Mann-Whitney U tests, and our statistical results indicated that there were no significant differences between PS and NP conditions on their basic pre-practice knowledge ($U = 1008.0$, $p = .846$, $CLES = .51$) or general CS self-efficacy ($U = 1146.0$, $p = .182$, $CLES = .58$), suggesting that the condition groups were comparable. We also used Cronbach's α reliability test to check the internal consistency of the survey questions, resulting in $\alpha = 0.82$ for the general CS self-efficacy survey and $\alpha = 0.80$ for the pre-practice knowledge in Python `class` survey, demonstrating that these two surveys had good internal consistency [20].

We calculated each student's in-practice problem-solving efficiency using the likelihood model from Hoffman and Schraw [23]. A learner would have a bigger relative gain and be seen as more efficient if they spent less time while still achieving more problem-solving accuracy [23]. Following this model, for each student, the in-practice problem-solving efficiency was calculated as the ratio of practice score ($Max = 40$) and practice time (mins). Practice time was calculated as the time used for practice, excluding any periods of inactivity over 5 minutes. The highest problem-solving efficiency was 4.07, achieved by a student who finished all four write-code problems (40 points) in 9.83 minutes.

To investigate how Parsons scaffolding may impact learners with varying CS self-efficacy levels differently, we divided learners into two groups based on their scores in the general CS self-efficacy survey. The PS-High ($N = 20$, $M = 25.1$, $SD = 2.7$) and NP-High ($N = 22$, $M = 23.8$, $SD = 2.7$) are students that scored higher on the CS self-efficacy survey; the PS-Low ($N = 21$, $M = 17.3$, $SD = 2.8$) and NP-Low ($N = 26$, $M = 16.2$, $SD = 2.4$) are students that scored lower in the general CS self-efficacy survey.

We found no differences between PS-High & NP-High groups in terms of the self-evaluated prior knowledge in Python `class` ($U = 230.0$, $p = .810$, $CLES = .52$) and PS-Low & NP-Low groups ($U = 275.5$, $p = .965$, $CLES = .50$). We then conducted a series of analyses to understand the differences in terms of practice performance, problem-solving efficiency, and posttest performance among the groups. In cases where the data was not normally distributed, we used the Mann-Whitney U test instead of ANOVA. Results are included in Table 5. We observed significant differences between PS-Low and NP-Low groups on write-code practice performance and problem-solving efficiency; PS-Low students had significantly higher write-code practice performance and problem-solving efficiency than NP-Low students. However, there is no significant difference in write-code practice performance and problem-solving efficiency between students in PS-High and NP-High groups. In addition, there are no significant differences in posttest performance between the two conditions for students in both low and high CS self-efficacy groups.

Table 3: Four Write-code Practice Problems

Problem Name	Question Item
1 - <i>Song</i>	Write a class <code>Song</code> with an <code>__init__</code> method that takes a <code>title</code> as a string and <code>len</code> as a number and initializes these attributes in the current object. Then define the <code>__str__</code> method to return the <code>title</code> , <code>len</code> . For example, <code>print(s)</code> when <code>s = Song('Respect', 150)</code> would print "Respect, 150".
2 - <i>Cat</i>	Write a class <code>Cat</code> with an <code>__init__</code> method that takes <code>name</code> as a string and <code>age</code> as a number and initializes these attributes in the current object. Next create the <code>__str__</code> method that returns "name: name, age: age". For example if <code>c = Cat("Fluffy", 3)</code> then <code>print(c)</code> should print "name: Fluffy, age: 3". Then define the <code>make_sound</code> method to return "Meow".
3 - <i>Account</i>	Create a class <code>Account</code> with an <code>__init__</code> method that takes <code>id</code> and <code>balance</code> as numbers. Then create a <code>__str__</code> method that returns "id, balance". Next create a <code>deposit</code> method takes <code>amount</code> as a number and adds that to the balance. For example, if <code>a = Account(32, 100)</code> and <code>a.deposit(50)</code> is executed, <code>print(a)</code> should print "32, 150".
4 - <i>FortuneTeller</i>	Write a class <code>FortuneTeller</code> with an <code>__init__</code> method that takes a list of fortunes as strings and saves that as an attribute. Then create a <code>tell_fortune</code> method that returns one of the fortunes in the list at random.

Table 4: Pre-practice knowledge level and general CS self-efficacy level by condition, reported in *M (SD), Mdn (25th percentile - 75th percentile)* format

Category (Score range)	PS (N=41)	NP (N=48)
pre-practice knowledge in Python <code>class</code> (4-20)	9.3 (4.9), 8.0 (4.0-13.0)	8.8 (4.3), 7.5 (5.8-12.0)
general CS self-efficacy level (6-30)	21.1 (4.8), 21.0 (18.0-24.0)	19.7 (4.6), 19.0 (17.0-23.0)

Table 5: Comparison of practice performance, problem-solving efficiency, and posttest performance by condition at high and low CS self-efficacy levels, reported in *Mdn (25th percentile - 75th percentile)* format

Category	Parsons Scaffolding - High	No Scaffolding - High	Statistical Results
Write-code practice	20.0 (10.0-40.0)	22.5 (0.0-35.0)	$U = 254.5, p = .381, CLES = .58$
Problem-solving efficiency	1.5 (0.7-2.4)	1.2 (0-2.7)	$U = 242.0, p = .585, CLES = .55$
Posttest performance	10.0 (0-36.7)	25.0 (0-36.2)	$U = 211.5, p = .837, CLES = .48$
Category	Parsons Scaffolding - Low	No Scaffolding - Low	Statistical Results
Write-code practice	20.0 (10.0-40.0)	0 (0-0)	$U = 442.5, p < .001, CLES = .81$
Problem-solving efficiency	1.3 (0.5-1.7)	0 (0-0)	$U = 435.0, p < .001, CLES = .80$
Posttest performance	0 (0-30.0)	0 (0-5.3)	$U = 335.0, p = .138, CLES = .61$

4.2 RQ2: In the Parsons Problems as Scaffolding (PS) condition, how did students with varying CS self-efficacy levels use the Parsons scaffolding?

In order to answer this RQ, we first need to describe the expected behavior of utilizing Parsons problems as scaffolding. Since the Parsons problem is optional and the scaffolding is supposed to be initiated by the students, we did not expect every student to use the Parsons problem to solve every write-code problem; we wanted them to use it when they were in need of help. In addition, as previously described in Section 2.5 and Section 3, students had the autonomy to engage with the Parsons problem in any way they chose. However, the different ways were interdependent because students were only able to interact with Parsons scaffolding in one specific way for each question. Therefore, to avoid redundant analyses, we selected the most extensive use of Parsons scaffolding (solve) to get a sense of the relationship between students' overall

CS self-efficacy and their utilization of Parsons scaffolding. We calculated *solve Parsons scaffolding rate* as the total number of times they solved Parsons scaffolding divided by the number of practice problems, which is four in our study. The mean (*M*) was 35%, with a standard deviation (*SD*) of 38%, ranging from 0% to 100%. We then computed a Pearson correlation and found a significant negative correlation between students' general CS self-efficacy and solve Parsons scaffolding rate, $r = -.32, p = .043$. Specifically, when given Parsons problems as scaffolding during practice, students with lower CS self-efficacy were more likely to solve them. In other words, students with higher CS self-efficacy tended to use Parsons scaffolding more lightly, or even solve problems independently.

Given that the ultimate goal of providing scaffolding is to help students solve write-code problems, we are also interested in knowing the relationship between their CS self-efficacy levels and how they used Parsons scaffolding to solve write-code problems. As a result, for each student, we then computed the *effective scaffolding rate* as the frequency of using the Parsons scaffolding in any of

the three methods and completing the corresponding write-code problem divided by the number of times they used the Parsons scaffolding. A high effective scaffolding rate (close to 1) indicates that the scaffolding was more effective in helping the students finish the write-code practice, while a low rate (close to 0) indicates the student did not benefit from the scaffolding as much. For instance, if a student used the Parsons problem for three write-code practice problems but only finished one write-code problem after using the scaffolding, then the effective scaffolding rate would be 0.33. The mean and standard deviation of the effective scaffolding rate is $M = 51\%$, $SD = 40\%$. We then computed the Pearson correlation between the effective scaffolding rate and students' general CS self-efficacy scales, which is $r = .18$, $p = .292$.

Posthoc Analysis Considering that the average effective scaffolding rate is 51%, this suggests that some Parsons scaffolding usage did not reach the expected outcome in helping students solve the write code problem. To better understand why and how some PS students used Parsons scaffolding less effectively, we conducted two follow-up analyses. We firstly computed the Pearson correlation between students' effective scaffolding rate and students' pre-practice knowledge level on Python `class`, and observed a significant positive relationship, $r = .40$, $p = .014$.

In addition, for those who utilized the Parsons scaffolding but still got the write-code practice wrong, we looked deeper into their final write-code submissions and the corresponding final Parsons scaffolding problem status. This resulted in a total of 47 Parsons scaffolding state & write-code state pairs. We classified these pairs into four categories based on the final Parsons scaffolding state, followed by the corresponding final code submission state: (1) *Only scanned Parsons blocks* (26 instances) - twenty-four (92%) of their final write code submissions did not compile due to syntax errors, type errors, or name errors, one student did not write any code after viewing Parsons blocks, and one student's code compiled successfully but had logic errors and did not pass all the unit tests. (2) *Completed Parsons problems but failed write-code task* (15 instances) - twelve of them had code in the write-code box, and three left the write-code box blank. In all 12 cases, students did the final write-code submission after getting the Parsons solution but still failed to solve it. Since they already had the correct solution in hand (the Parsons solution), we examined their actual code to understand the specific errors they made better. Our results showed that, while these students completed the Parsons scaffolding problem, they still might not have acquired enough skill in writing code on this topic. For example, in six cases, students received a type error by writing `_init_ or _str_`, which should be `_init_ or _str_`. And two students had syntax errors in the longest line of `Song`: `return self.title + ", " + str(self.len)`, such as missing the `+` or miswriting as `str().self.len`. We also found one student incorrectly placed the correct solution after the default test cases for three problems. (3) *Incorrect Parsons completion* (2 instances) - both of them omitted some problem requirements and did not finish the code. For example, one student did not complete the required `def deposit(self, amount)` by leaving this part blank; (4) *Attempted to move Parsons blocks but could not finish* (4 instances) - two final code submissions did not complete the requirements, and two did not write any code.

In summary, we found that students with lower levels of general CS self-efficacy were more likely to solve Parsons problems as scaffolding during practice. Furthermore, when choosing to use Parsons scaffolding, students with higher pre-practice knowledge in Python `class` were more likely to use it effectively. However, the effectiveness was not related to students' CS self-efficacy levels. Our preliminary analysis of students' code submissions revealed that the current Parsons scaffolding method might still be too challenging for some students.

4.3 RQ3: In the Parsons problems as Scaffolding (PS) condition, how did students rate the usefulness of the Parsons scaffolding and why?

To get a general sense of how useful this scaffolding method was, we first looked at students' ratings on the usefulness of the Parsons problems as scaffolding. If students did not apply the scaffolding for the write-code question, they were told to skip the related rating question. The distribution of student ratings for each problem is shown in Figure 4. We found that for all four problems, nearly 70% of the ratings were five or more. This indicates that students generally found Parsons problems helpful in solving the write-code practice problems, which target new concepts they are learning. In addition, 71% (29 out of 41) of students in the PS condition completed the open-ended question to explain their ratings. Specifically, 19 (66%) of them explained how Parsons scaffolding helped them while two of them reported specific challenges they faced, and the rest of the eight answers (28%) explained their ratings by discussing how difficult this practice was for them.

The average usefulness rating for those 19 students was 7.7, indicating that those students found Parsons scaffolding to be helpful. Of those 19 positive explanations, two responses only stated that Parsons problems were helpful without further elaboration. For the rest of the answers, four students (21%) stated that Parsons problem made it easier for them to solve the write-code problem, but still learn. For example, one student wrote *"The Parsons problem gives me all the necessary elements of creating a class which I am unfamiliar with, but I still have to figure out an order which is helping me learn, but not too strenuous"*. Four students (21%) thought Parsons problem helped them learn problem-solving strategies, as one student explained, *"I feel like Parsons helped me understand the different pieces of code and how to think about the problems."* A high proportion of students (47.38%, 9 out of 19 students) valued the benefit of refining and extending existing programming knowledge by using Parsons scaffolding. One student claimed that *"I am still confused about class, method, and self, but these problems definitely helped me understand better"*, another one expressed a similar idea, *"the Parsons problems helped me remember how to create methods and enter arguments as necessary."* Additionally, one student elaborated on how the Parsons scaffolding helped to extend the existing programming knowledge in a more detailed way, *"I understood structurally what I had to do; I just forgot how to return one of the fortunes in the list at random, so the Parsons problem helped me figure out how to accomplish that."*

However, some participants reported difficulties when using Parsons scaffolding to complete write-code problems, as evidenced

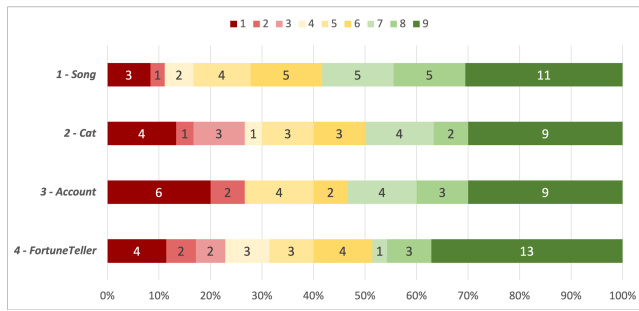


Figure 4: Stacked bar chart of Parsons scaffolding usefulness ratings for each practice problem

by their low usefulness ratings and corresponding negative explanations. For instance, the student who rated *Cat* three and *FortuneTeller* one did not finish those two problems and explained the low ratings further as *"I still do not understand class and found this exercise extremely frustrating ..."* Similarly, the student who rated *Account* one only scanned the Parsons problem and did not finish the write-code problem. This student emphasized the inherent high difficulty as *"This was really hard to just dive right into."* As mentioned above, we also had two students who explicitly explained why Parsons scaffolding was less useful for them. One of the two students used the Parsons scaffolding four times but was unable to finish any of the associated write-code problems. This student gave an average usefulness rating of 4.8 and provided the following explanation: *"I think it helps me get started, but because this is completely new to me, it merely helps me see what is there, not how to put it together."* Another student received an effective scaffolding rate of 0.5 and reported an average usefulness rating of 4.3. This student explained the rating further as *"... Parsons isn't really that helpful because you're just copying."*

5 DISCUSSION

This study investigates whether and how using Parsons problems can scaffold the write-code practice with students of different CS self-efficacy levels. We found that, for newly learned difficult programming concepts, when having Parsons problem as write-code scaffolding, students with low CS self-efficacy (PS-Low) achieved significantly higher practice performance and problem-solving efficiency than those with low CS self-efficacy in the no scaffolding condition (NP-Low). However, this condition effect did not occur in students with high CS self-efficacy (PS-High and NP-High). Also, there were no condition effects on posttest performance among the groups.

Beyond intervention effects, for PS students, we also examined the relationship between their general CS self-efficacy and the use of Parsons scaffolding. We discovered that students with lower levels of CS self-efficacy are more likely to solve Parsons problems as scaffolding during practice. To better understand the ineffective scaffolding use, we also conducted two follow-up analyses and found that, when choosing to use the Parsons scaffolding, students with higher pre-practice knowledge in Python `class` were more likely to use it effectively; however, this was not related to students' general CS self-efficacy. Further, our preliminary analysis of student

code found that this Parsons scaffolding method might still be too challenging for some students. After analyzing students' useful ratings and explanations, we found Parsons scaffolding could help students to learn to write code, but there are still areas to improve. In this section, we will discuss our findings regarding the condition effects for students with high and low CS self-efficacy levels (RQ1). In the next section, we will present some design implications based on the results of RQ2 and RQ3.

5.1 RQ1: Parsons scaffolding is more beneficial for students with low CS self-efficacy in terms of practice performance and in-practice problem-solving efficiency

According to our results, for students with low general CS self-efficacy, those who received Parsons scaffolding (PS-Low) achieved significantly higher practice performance and in-practice problem-solving efficiency compared to those who did not receive scaffolding (NP-Low). However, we did not obtain such condition effects between students with high CS self-efficacy (PS-High and NP-High). This could be explained by the idea that a person with a higher level of self-efficacy will exert more effort on learning tasks and will persevere when confronted with difficulties [6, 52, 57]. Given that the programming concept we used in this study was relatively new to students, most of them had little prior experience writing code about it except for the three examples of code they could execute in the instruction phase. However, students with high self-efficacy in computer science, regardless of whether they received Parsons scaffolding or not, were likely more willing to continue with this frustrating process of writing code. Therefore, they achieved a similar level of practice performance and problem-solving efficiency during practice.

On the other hand, when there was no scaffolding, solving those write-code problems may seem hopeless for students with low CS self-efficacy. Therefore, it was common for NP-Low students to give up quickly rather than persevere in overcoming the difficulties. Nevertheless, although PS-Low students were also prone to giving up, Parsons problems were there to assist them in solving write-code problems that they would not have been able to solve on their own. Consequently, by providing a more supportive learning experience, students with lower CS self-efficacy could finish more write-code practice and achieve higher in-practice problem-solving efficiency.

As for posttest performance, we found no significant differences among the groups. This outcome can be explained by the inherent high difficulty of writing code for a newly learned concept from scratch. Although PS students practiced more successfully than NP students during practice in general, it may not have been enough for them to become proficient in code writing with `class` because of the small number of practice problems we applied in this study [1]. In the future, we could use student models to track their write-code skill development and determine the number of practice problems they require to master this topic [8].

5.2 RQ2 & RQ3: Improve the effectiveness of using Parsons problems to scaffold write-code exercise

Our result from RQ2 indicated that students with lower CS self-efficacy were more likely to solve the Parsons problem when they had the scaffolding. One possible explanation is that learners with lower CS self-efficacy were less confident in their ability to perform the write-code problem successfully, similar to Wang et al. [52]. As a result, they were more willing to solve a partially correct solution (Parsons problem) and follow it. However, students with higher CS self-efficacy levels were more confident in completing the write-code exercise. They would rather spend more time on their own code and, if necessary, use Parsons scaffolding in a more limited way. In addition, we also found that not all the Parsons scaffolding usage was effective. We found that students with a lower pre-practice knowledge level are more likely to use Parsons scaffolding ineffectually. Our analysis of their code also revealed several ineffective scenarios, which led to design suggestions for different groups of students.

Firstly, one possible reason why some students struggle with using Parsons scaffolding is that it may still be too difficult. Although we implemented a "Help Me" button for student-initiated difficulty level adaptation, students still had to complete at least three full attempts to activate it. It was possible that students became overwhelmed by the default multiple Parsons blocks, causing them to give up before even attempting three times. Therefore, an important enhancement of the current scaffolding approach would be providing the Parsons problems with appropriate levels of difficulty based on students' submission histories or incorrect code. In addition, to prevent students from feeling overwhelmed by encountering a new-looking problem (the equivalent Parsons problem) other than the write-code problem, it is important to establish a connection between the Parsons scaffolding and their existing code. One way to achieve this is to personalize the Parsons problem. Instead of providing the most common previous student solution, we could create a Parsons problem that is based on the student's incorrect code. Furthermore, to give students a sense of accomplishment and relatedness between existing written code and Parsons scaffolding, we could make the correct parts of the existing student code static with positive feedback [34].

Secondly, we found that some students finished the Parsons problem as scaffolding but still had errors in their write-code submissions. One potential reason is that, while these students finished the Parsons scaffolding problem, they were unable to localize their errors, such as the double leading and trailing underscores in Python, and could not use the Parsons solution effectively. Alternatively, students might prefer not to follow the Parsons solution, and therefore keep the errors in their own approach. These call for a more personalized Parsons scaffolding approach, which includes using solutions close to the students' current path and adding paired distractors that can highlight errors based on the student's current code state. Besides refining the scaffolding mechanism, incorporating explanations is another way to boost the efficiency of Parsons scaffolding. One direction to achieve this is to provide prompt guidance to make learners' self-explanation on Parsons scaffoldings

more productive since it helps them concentrate on relevant information [32, 39]. Another direction is adding textual explanations to either Parsons blocks or the finished Parsons solution. A prior study by Marwan et al. added explanations to next-step hints and found that novices thought hints with explanations were much more relevant and understandable [35]. They were also better able to relate these hints to their code. We believe this method could improve students' effective scaffolding rate.

Moreover, given that we found some students finished the Parsons scaffolding and left the write-code box blank, it is possible that those students felt it was unnecessary to retype the correct Parsons solution into the write-code box. While we think that the process of retyping might enrich their comprehension and help them to identify some points that they might not have noticed without typing on their own, we could add an "AutoFill" button for those who made considerable progress when utilizing the Parsons scaffolding.

6 LIMITATION AND FUTURE WORK

One limitation of this work is that we did not collect subjective ratings or perceptions from participants who did not use Parsons problems as scaffolding. We plan to add some concrete survey questions to learn more about why they chose not to use the scaffolding. In addition, future retrospective interviews are necessary to fully understand the ineffective scaffolding cases. Moreover, we only conducted this study on one topic in a medium-scale classroom at one public university in the United States. With other demographic groups, computing domains, and educational settings, like data science and MOOCs, we might observe other Parsons usage scenarios and scaffolding effects. Furthermore, in order to save class time and reduce pretest cognitive overload for students, we only utilized self-reported measures to assess their pre-practice knowledge level in Python `class`, which may be subject to bias.

Regarding future work, we would like to investigate the impact of Parsons problems as scaffolding with other scaffolding techniques, in other programming languages, and educational settings. In addition, we are excited to continue our work based on the design suggestions provided above, such as providing personalized Parsons problems that are based on the student's existing incorrect solution and adapting the Parsons problem scaffolding by setting the starting state of the Parsons problem as the final state of the student's code. In addition, we also look forward to reducing the cost of developing Parsons problems by automating the process based on student submissions and large language models.

7 CONCLUSION

In this work, we investigated the effects of Parsons problems as scaffolding during code writing skill acquisition for students of various CS self-efficacy levels. We found that lower CS self-efficacy students in the Parsons as scaffolding condition achieved significantly higher practice performance and problem-solving efficiency than lower CS self-efficacy students in the condition without any scaffolding. Further investigations into student interaction with the Parsons scaffolding revealed that students with lower levels of CS self-efficacy are more likely to solve Parsons scaffolding problems

during practice. In addition, when choosing to use Parsons problems as scaffolding, students with higher pre-practice knowledge of the topic were more likely to use them effectively; however, this was not related to students' general CS self-efficacy. These findings direct us to optimize the Parsons scaffolding experience, including providing personalized and adaptive versions of the Parsons problems based on the student's current problem-solving status.

ACKNOWLEDGMENTS

The funding for this research came from the National Science Foundation award 2143028. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

- [1] Robert K Atkinson, Sharon J Derry, Alexander Renkl, and Donald Wortham. 2000. Learning from examples: Instructional principles from the worked examples research. *Review of educational research* 70, 2 (2000), 181–214.
- [2] Albert Bandura. 1977. Self-efficacy: toward a unifying theory of behavioral change. *Psychological review* 84, 2 (1977), 191.
- [3] Albert Bandura and Richard H Walters. 1977. *Social learning theory*. Vol. 1. Englewood cliffs Prentice Hall.
- [4] João Henrique Berssanette and Antonio Carlos de Francisco. 2021. Cognitive Load Theory in the Context of Teaching and Learning Computer Programming: A Systematic Literature Review. *IEEE Transactions on Education* (2021).
- [5] Benjamin S Bloom. 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher* 13, 6 (1984), 4–16.
- [6] Shari L Britner and Frank Pajares. 2006. Sources of science self-efficacy beliefs of middle school students. *Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching* 43, 5 (2006), 485–499.
- [7] Ünal Çakıroğlu and Dilara Arzgül Aksoy. 2017. Exploring extraneous cognitive load in an instructional process via the web conferencing system. *Behaviour & Information Technology* 36, 7 (2017), 713–725.
- [8] Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction* 4, 4 (1994), 253–278.
- [9] 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*. 113–124.
- [10] Paul Denny, James Prather, Brett A Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A closer look at metacognitive scaffolding: Solving test cases before programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 1–10.
- [11] Rodrigo Duran, Jan-Mikael Rybicki, Juha Sorva, and Arto Hellas. 2019. Exploring the value of student self-evaluation in introductory programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 121–130.
- [12] Barbara Ericson, Austin McCall, and Kathryn Cunningham. 2019. Investigating the affect and effect of adaptive parsons problems. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 1–10.
- [13] Barbara J Ericson, Paul Denny, James Prather, Rodrigo Duran, Arto Hellas, Juho Leinonen, Craig S Miller, Briana B Morrison, Janice L Pearce, and Susan H Rodger. 2022. Parsons problems and beyond: Systematic literature review and empirical study designs. *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (2022), 191–234.
- [14] 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*. 60–68.
- [15] 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*. 169–178.
- [16] 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*. 20–29.
- [17] Barbara J Ericson and Bradley N Miller. 2020. Runestone: A platform for free, online, and interactive ebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 1012–1018.
- [18] Rita Garcia. 2021. Evaluating Parsons Problems as a Design-Based Intervention. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.
- [19] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.
- [20] Joseph A Gliem and Rosemary R Gliem. 2003. Calculating, interpreting, and reporting Cronbach's alpha reliability coefficient for Likert-type scales. Midwest Research-to-Practice Conference in Adult, Continuing, and Community
- [21] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 599–608.
- [22] Carl C Haynes and Barbara J Ericson. 2021. Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.
- [23] Bobby Hoffman and Gregory Schraw. 2010. Conceptions of efficiency: Applications in learning and problem solving. *Educational Psychologist* 45, 1 (2010), 1–14.
- [24] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2022. Using Adaptive Parsons Problems to Scaffold Write-Code Problems. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 15–26.
- [25] Petri Ihanntola and Ville Karavirta. 2011. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education* 10, 2 (2011), 119–132.
- [26] Minchi C Kim and Michael J Hannafin. 2011. Scaffolding problem solving in technology-enhanced learning environments (TELEs): Bridging research and theory with practice. *Computers & Education* 56, 2 (2011), 403–417.
- [27] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.
- [28] Charles B Kreitzberg and Len Swanson. 1974. A cognitive model for structuring an introductory programming curriculum. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*. 307–311.
- [29] Colleen M Lewis, Ken Yasuhara, and Ruth E Anderson. 2011. Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In *Proceedings of the seventh international workshop on Computing education research*. 3–10.
- [30] Raymond Lister. 2020. On the cognitive development of the novice programmer: and the development of a computing education researcher. In *Proceedings of the 9th Computer Science Education Research Conference*. 1–15.
- [31] Andrew Luxton-Reilly, Simon, Ibrahim Alblui, Brett A Becker, Michail Gianakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, et al. 2018. Introductory programming: a systematic literature review. In *Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education*. 55–106.
- [32] Lauren E Margulieux and Richard Catrambone. 2019. Finding the best types of guidance for constructing self-explanations of subgoals in programming. *Journal of the Learning Sciences* 28, 1 (2019), 108–151.
- [33] Samiha Marwan, Anay Dombe, and Thomas W Price. 2020. Unproductive help-seeking in programming: What it is and how to address it. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 54–60.
- [34] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W Price, and Tiffany Barnes. 2020. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM conference on international computing education research*. 194–203.
- [35] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proceedings of the 2019 ACM conference on innovation and technology in computer science education*. 520–526.
- [36] Richard E Mayer and Roxana Moreno. 2003. Nine ways to reduce cognitive load in multimedia learning. *Educational psychologist* 38, 1 (2003), 43–52.
- [37] Irene T Miura. 1987. The relationship of computer self-efficacy expectations to computer interest and course enrollment in college. *Sex roles* 16, 5 (1987), 303–311.
- [38] 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*. 42–47.
- [39] Huy A Nguyen, Xinying Hou, Hayden Stec, Sarah Di, John Stamper, and Bruce M McLaren. 2023. Examining the Learning Benefits of Different Types of Prompted Self-explanation in a Decimal Learning Game. In *International Conference on Artificial Intelligence in Education*. Springer, 681–687.
- [40] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive load theory and instructional design: Recent developments. *Educational psychologist* 38, 1 (2003), 1–4.
- [41] 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. 157–163.
- [42] James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM conference on international computing education research*. 2–13.
- [43] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education* 29, 3 (2019), 368–395.
- [44] Alexander Renkl, Robert K Atkinson, and Cornelia S Große. 2004. How fading worked solution steps works—a cognitive load perspective. *Instructional science* 32, 1-2 (2004), 59–82.
- [45] Kelly Rivers. 2017. *Automated data-driven hint generation for learning programming*. Ph. D. Dissertation. Carnegie Mellon University.
- [46] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educational psychology review* 19, 4 (2007), 469–508.
- [47] Sue Sentance and Andrew Cszizmadia. 2017. Computing in the curriculum: Challenges and strategies from a teacher’s perspective. *Education and Information Technologies* 22, 2 (2017), 469–495.
- [48] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review* 22, 2 (2010), 123–138.
- [49] John Sweller and Paul Chandler. 1994. Why some material is difficult to learn. *Cognition and instruction* 12, 3 (1994), 185–233.
- [50] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31 (2019), 261–292.
- [51] Jeroen JG Van Merriënboer and John Sweller. 2005. Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review* (2005), 147–177.
- [52] Jiahui Wang, Abigail Stebbins, and Richard E Ferdig. 2022. Examining the effects of students’ self-efficacy and prior knowledge on learning and visual behavior in a physics game. *Computers & Education* 178 (2022), 104405.
- [53] Nathaniel Weinman, Armando Fox, and Marti A Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–4.
- [54] Eric Wiebe, Laurie Ann Williams, Kai Yang, and Carol S Miller. 2003. *Computer science attitude survey*. Technical Report. North Carolina State University. Dept. of Computer Science.
- [55] Susan Wiedenbeck. 2005. Factors affecting the success of non-majors in learning to program. In *Proceedings of the first international workshop on Computing education research*. 13–24.
- [56] Joseph B Wiggins, Joseph F Grafsgaard, Kristy Elizabeth Boyer, Eric N Wiebe, and James C Lester. 2017. Do you think you can? the influence of student self-efficacy on the effectiveness of tutorial dialogue for computer science. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 130–153.
- [57] Amy L Zeldin and Frank Pajares. 2000. Against the odds: Self-efficacy beliefs of women in mathematical, scientific, and technological careers. *American educational research journal* 37, 1 (2000), 215–246.