Parsons Problems and Beyond: Systematic Literature Review and Empirical Study Designs

Barbara J. Ericson[*]
barbarer@umich.edu
University of Michigan
Ann Arbor, MI, USA

Paul Denny[*]
p.denny@auckland.ac.nz
The University of Auckland
Auckland, New Zealand

James Prather[*]
james.prather@acu.edu
Abilene Christian University
Abilene, Texas, USA

Rodrigo Duran
rodrigo.duran@ifms.edu.br
Fed. Institute of Mato Grosso do Sul
Nova Andradina, Brazil

Arto Hellas
arto.hellas@aalto.fi
Aalto University
Espoo, Finland

Juho Leinonen
juho.2.leinonen@aalto.fi
Aalto University
Espoo, Finland

Craig S. Miller
craig.miller@depaul.edu
DePaul University
Chicago, Illinois, USA

Briana B. Morrison
bbmorrison@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Janice L. Pearce
jan_pearce@berea.edu
Berea College
Berea, Kentucky, USA

Susan H. Rodger
rodger@cs.duke.edu
Duke University
Durham, North Carolina, USA

ABSTRACT

Programming is a complex task that requires the development of many skills including knowledge of syntax, problem decomposition, algorithm development, and debugging. Code-writing activities are commonly used to help students develop these skills, but the difficulty of writing code from a blank page can overwhelm many novices. Parsons problems offer a simpler alternative to writing code by providing scrambled code blocks that must be placed in the correct order to solve a problem. In the 16 years since their introduction to the computing education community, an expansive body of literature has emerged that documents a range of tools, novel problem variations and makes numerous claims of benefits to learners. In this work, we track the origins of Parsons problems, outline their defining characteristics, and conduct a comprehensive review of the literature to document the evidence of benefits to learners and to identify gaps that require exploration. To facilitate future work, we design empirical studies and develop associated resources that are ready for deployment at a large scale. Collectively, this review and the provided experimental resources will serve as a focal point for researchers interested in advancing our understanding of Parsons problems and their benefits to learners.

CCS CONCEPTS

• **Social and professional topics** → *Computing education.*

KEYWORDS

Parsons Problems, Parsons Puzzles, Parson's Programming Puzzles, Parson's Problems, Parson's Puzzles, Code Puzzles

ACM Reference Format:
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. In *2022 ITiCSE Working Group Reports (ITiCSE-WGR '22), July 8–13, 2022, Dublin, Ireland*. ACM, New York, NY, USA, 44 pages. https://doi.org/10.1145/3571785.3574127

1 INTRODUCTION

Programming requires mastering a number of distinct but prerequisite skills, which presents challenges for both learners and teachers. Students must develop familiarity with language syntax and constructs, learn how to decompose problems and design algorithms, identify and apply useful programming patterns, and develop an ability to read and comprehend code at different levels of complexity [144, 228]. Programming courses often expect students to learn these skills from writing code [92], but the complexity of this task can overwhelm novice students [124]. Computing educators and researchers have thus spent significant effort exploring a wide range of pedagogical approaches and activities to help students develop these necessary skills [18].

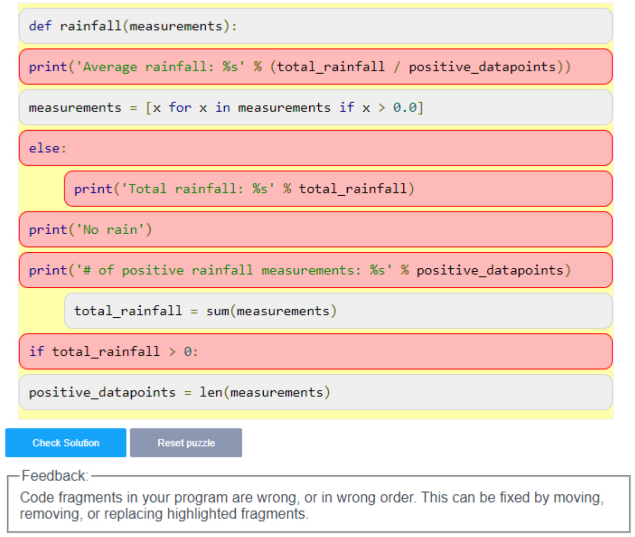
One such activity, originally introduced by Dale Parsons and Patricia Haden in 2006, aimed to provide an engaging puzzle-based experience that allowed for the modelling of good code and that facilitated immediate feedback [177]. So called "Parsons problems"

[*] co-leader

ITiCSE-WGR '22, July 8–13, 2022, Dublin, Ireland
© 2022 Association for Computing Machinery.
ACM ISBN 979-8-4007-0010-1/22/07…$15.00
https://doi.org/10.1145/3571785.3574127

presented the lines of code comprising a solution in a scrambled order and tasked students with rearranging them into the correct order. Compared to writing code into an initially blank editor, which can often be overwhelming for novices, Parsons problems greatly constrain the problem-solving space while at the same time expose students to syntax and common code structures and patterns. Thus, they offer a simpler, scaffolded alternative to more authentic code writing tasks. Figure 1 shows an example of a typical Parsons problem in Python, created using the editor by Codio[1] which builds on the popular js-parsons tool (introduced in [109]), where solving the puzzle entails dragging and dropping the blocks into the correct order and with the correct indentation. Line-based feedback, which highlights code blocks currently with the wrong relative order or indentation, is shown.

Since their initial introduction more than 16 years ago, Parsons problems have been widely adopted, adapted and studied. In particular, researchers have explored many novel variations of Parsons problems and have developed a wide variety of tools for their delivery [4, 11, 34, 82, 96, 106, 108, 109, 129, 225]. A large and growing body of work now documents their use, efficacy, and limitations in computing classrooms around the world. However, there have been no large scale efforts to organise this literature and no large scale attempts at replication to verify these claims. As their popularity continues to grow, we see a pressing need for a comprehensive review of research on Parsons problems to understand how they have been studied in educational settings, what type of evidence has been presented for their effectiveness, and what gaps exist. This work will be of benefit to researchers, in helping them chart and focus future research directions, as well as to educators who are looking to adopt effective pedagogical approaches in the classroom.



```
def rainfall(measurements):
print('Average rainfall: %s' % (total_rainfall / positive_datapoints))
measurements = [x for x in measurements if x > 0.0]
else:
    print('Total rainfall: %s' % total_rainfall)
print('No rain')
print('# of positive rainfall measurements: %s' % positive_datapoints)
    total_rainfall = sum(measurements)
if total_rainfall > 0:
positive_datapoints = len(measurements)
```

Feedback:
Code fragments in your program are wrong, or in wrong order. This can be fixed by moving, removing, or replacing highlighted fragments.

Figure 1: Example Parsons problem code blocks for the classic 'Rainfall' problem, showing feedback on line ordering

In this report, we explore the origins and theoretical underpinnings of Parsons problems, including an interview with Dale Parsons, and we document their defining characteristics. This provides a framework that serves to capture the many variations of Parsons problems that have been proposed and deployed. We then conduct a rigorous review of the literature, involving examination of more than 1,000 articles from different sources, and explore the research contexts and problem features that have been studied, the specific questions that have driven the existing research, and the quality and limitations of the evidence put forth in support of the benefits to learners. Through this review, we identify and present the main gaps that exist in the literature with respect to the contexts, research themes, and evidence presented, and we publish a list of research questions that require further exploration. As research interest in Parsons problems and their many variations continues to grow, our aim is that this review will serve as a blueprint for future research efforts. To facilitate work that can address some of these shortcomings, we have selected a subset of the open research questions and designed materials and protocols for their investigation which we have refined through small-scale pilot studies. We propose a replicable study design, provide associated resources that are ready for immediate use, and invite the broader computing education community to make use of these experimental resources to explore the efficacy of Parsons problems on a multi-institutional and multi-national scale.

1.1 Report structure

In Section 2, we investigate the origins of Parsons problems and explore their defining characteristics. We report the results of an interview with Dale Parsons (the full transcript of which appears in Appendix B), review background literature both within and outside of computing, and examine several related theories that underpin and help to explain their educational benefits. In Section 3, we define the research questions that drove our review of the literature, and we carefully describe our approach to this review, including validation of our chosen search terms and constructing our inclusion and exclusion criteria. The results of the literature review are then presented in Section 4, organised around our primary research questions. We classify the contexts in which studies have been conducted and the features and variations of Parsons problems that have been investigated, thematically analyse the research questions explored by the reviewed literature, and critically analyse the evidence that has been documented for the benefits of Parsons problems. In Section 5, we identify gaps that emerged from the literature review and highlight promising research directions. In Section 6, we describe the experimental resources and protocols we have developed and for which we invite community engagement. These resources target important questions where further research is needed at scale, including investigation of the effects of solving Parsons problems with and without distractors, and exploring the benefits and challenges of solving Parsons problems compared with writing the equivalent code. To simplify the multi-institutional, multi-national deployment of these experimental resources, we leverage the open-source ebook, Runestone [63], which is a robust

[1] https://codio.github.io/parsons-puzzle-ui/

platform for delivery of Parsons problems and for running experiments at scale. Finally, the report concludes with a discussion of future directions in Section 7.

2 DEFINING PARSONS PROBLEMS

We begin this section by providing some historical context, initially through a conversation with the eponymous author of the original paper and then by considering similar kinds of rearrangement tasks and their uses in education and beyond. We then propose a taxonomy with respect to the typical characteristics of Parsons problems which can be used to describe their many variations and to differentiate them from other educational activities. Finally, we highlight how our work differs from an earlier literature review on the topic, and we outline several key learning theories that commonly underpin Parsons-focused research.

2.1 Origins

The origin of Parsons problems, as they are now commonly named, can be traced back to the 2006 paper [177] published by Dale Parsons and Patricia Haden from Otago Polytechnic, New Zealand. The paper, *'Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses',* was presented at the Australasian Computing Education (ACE) conference, held that year in Hobart, Australia. In this seminal work, Parsons and Haden acknowledged the importance of practice for mastering programming syntax and semantics, but they observed two problems with typical drill exercises. The first was that such problems tend to be boring, making it difficult for students to persist with practice. The second was the challenge of isolating the essential aspects of syntactic practice from the more conceptual problem solving elements of programming tasks. They proposed the idea of Parsons problems to help students memorise syntactic constructs while learning about algorithms and logical flow, and also to be engaging so that students are motivated to continue practicing. Overall, the design of these puzzles as described by Parsons and Haden adhered to five principles: maximise engagement, constrain the logic, permit common errors, model good code and provide immediate feedback.

Dale Parsons retired from Otago Polytechnic in 2021 after 30 years working as a lecturer in the School of Information Technology (in 2006, this was the School of Information Technology and Electrotechnology). To help us better understand the origins of Parsons problems, from the design of the activities to the motivations for publishing the work, Dale Parsons kindly agreed to be interviewed for this research. We recorded a thirty minute interview with her on 19th July, 2022.

We summarize below some of the main insights and highlights of the interview, and attach the full interview transcript in Appendix B.

(1) *The apostrophe* – There was never supposed to be an apostrophe in the name 'Parsons Programming Puzzles'. The original title was just 'Programming Puzzles', but then very late the night the paper was due, they changed the name to 'Parson's Programming Puzzles' using search-and-replace. This led to the accidental addition of an apostrophe to 'Parson's' that was not supposed to be there, but due to the late hour the mistake went unnoticed.
(2) *Alliteration* – Haden came up with the idea of adding "Parsons" to "Programming Puzzles" because she liked alliteration.
(3) *Partnership* – Parsons created the puzzles to use in her course, but Haden was the one that saw they would make a great paper, and this was the start of a perfect partnership for many years.
(4) *Errors* – Parsons created the puzzles based on common errors she had observed students making every year in the hope that the puzzles would help students recognize the errors.
(5) *Motivation* – The puzzles took off more with struggling students; they always wanted more of them. They were also motivating for students as students found them fun to do.
(6) *Timing* – Parsons would like to see Parsons Programming Puzzles introduced early on for novices. She was primarily using them post-lab and for assessments. She thought early introduction would be good because it might be difficult for a student to get a complete program written, but with the puzzles, they could feel they had achieved something and would feel better about their code.
(7) *Impact* – When Parsons and Haden presented the paper at ACE back in 2006, they had no idea at the time that their idea would gain such traction nor that others would build upon it.

2.2 Earlier Examples of Arrangement Tasks

Parsons problems involve the rearrangement of blocks, and are just one of many examples of arrangement-style tasks used in computing and education contexts. One of the earliest examples of arrangement tasks dates back to the concept of picture sequence arrangement used as part of intelligence tests. Bowler wrote an article in 1917 [26] discussing a picture arrangement test designed to measure logical judgments that was tested on 500 school children in Brussels by Dr. O. Decroly. This original test consisted of eleven series of pictures taken from children's books, where each series tell a complete simple story when arranged in the right order. A collection of pictures were given to the test subject (a child) in a random order who would then attempt to arrange them in a way that they tell a continuous story. The original goal was to find a series of pictures that adapted to different ages. The purpose of such tests was to approximately indicate the mental age of the test-taker.

The Picture Arrangement task was used in a series of intelligence tests for US Army recruits in World War I [232]. David Wechsler, who had administered psychological intelligence tests during World War I, borrowed the Picture Arrangement task (and others) to create the Wechsler scales of intelligence [224] in 1939. The Wechsler scales consisted of a collection of subtests, one of which was a picture-sequencing task. The picture-sequencing task existed until the Weschler-IV when it was dropped [13]. Since those original picture sequencing tasks used in intelligence tests, the idea of sequencing items has been adapted to multiple disciplines including Language Arts (storytelling), Mathematics (proofs), and Computer Science (Parsons problems).

In the broader context of computing, striving for order has been a prevalent objective. The research into sorting and searching algorithms was fundamental to the evolution of computer science, where algorithms such as merge sort changed the field. Sorting things was not just a task for the computer, but also for programmers. Whoever dropped a pile of punch cards would have to reorder them painstakingly back into the precise order for the program to run correctly. When working with punch cards, drafting programs on paper first was relatively common to verify the logic of the program before the "holy implementation". Ordering has been prevalent also in early post-punch card programming languages – as an example, the earliest versions of BASIC mandated the existence of line numbers, which programmers would then use to maintain the flow of the program.

The importance of ordering has been present also in computing education and teaching programming. Many of the working group members have recollections of having to sort paper snippets of code during their studies. Reaching out to a retired professor of one of the working group members who used such tasks in the 1980s to ask where the inspiration for creating paper-based code sorting tasks came from, resulted in the retired instructor responding with:

> *I guess I was inspired by the early research in constructing computer models of failed math problem-solving. (The researchers wanted to model the lattice of common failed solution paths so you could give meaningful coaching based on the student's position in the lattice.) It turns out that failure was mostly due to getting some necessary step out of proper order and not due to some conceptual failure. Right step at the wrong time or wrong step at the right time. That is, someone might have a perfectly good concept of "carry" when discussing the addition of a column of numbers, but if the student initiates the carry at the wrong place in the sequence, the solution fails. It's worse in CS, where one out-of-order step can destroy your whole world.* (Walter Maner, personal communication, May 8, 2022)

Recollections from other working group members included having Teaching Assistants (TAs) bring paper-based code fragments into classes, as well as creating such fragments themselves. The use of paper code fragments in teaching has been previously discussed in an article at least by Haatainen et al. [93], who used paper fragments in support sessions for struggling students.

2.2.1 *Outside CS.*

The idea of sequencing fragments of a solution are used in at least three other disciplines: Language Arts (in reading), Foreign Language Acquisition, and Mathematics. Language Arts adapts the original Picture Arrangement task by asking students to arrange pictures or story fragments into a temporal order [198]. There are also standardized assessments for this skill [103]. Interestingly, a recent line of research uses programming, specifically robots, to help teach sequencing to young students [121, 122].

Duolingo[2] is a tool for learning a foreign language. One type of puzzle it provides is to give a sentence in one language and scrambled words in another language, possibly with distractor words. The learner arranges selected words to form a sentence with the same meaning. The original sentence could be in either the language the learner already knows or the language the learner is learning.

Within Mathematics, the task of sequencing items most commonly occurs with proofs, specifically proofs in discrete math. There are two prominent tools that support students sequencing discrete math proof statements, MathsTiles [23–25] and Proof Blocks [180, 181].

MathsTiles was created as part of the Intelligent Book project and was originally a web-based homework tutor for discrete mathematics [23]. MathsTiles presents a block-based interface, similar to block-based programming editors, of composable tiles each of which can contain an arbitrary piece of mathematics or logic written by the teacher. Students then use the tiles to construct proofs in the area of number theory that are analysed with the Isabelle/HOL theorem prover [24]. MathsTiles allow students to have many answer fragments on the canvas at the same time and does not constrain the order in which a proof is written. Billingsley and Robinson [25] conducted a user study of MathsTiles where an introduction to the system and six proof exercises was made publicly available, and its use over three weeks in July 2006 was examined. A range of users were asked to try the system, including undergraduate students, postgraduate tutors of discrete mathematics, and other interested parties. While 83 people accessed the website, only 19 people accessed the introductory material, and only three people completed five of the six proofs (no one completed all six proofs). In these limited findings, users were only successful if they were provided a small instructor-procured subset of blocks.

Proof Blocks is implemented within PrairieLearn [213]. The original paper [180] describes the tool and its implementation. Proof Blocks is a tool which enables students to construct mathematical proofs by dragging and dropping prewritten proof lines into the correct order. Proof Blocks problems can be graded completely automatically. A key feature of Proof Blocks is in the specification of the correct answer, which allows for specifying a dependency graph of the lines of the proof, so that any correct arrangement of the lines can receive full credit. The follow-on paper [181] describes the use of Proof Blocks as exam questions and provides statistical evidence that Proof Blocks are easier than written proofs. The results also indicate that Proof Blocks problems provide about as much information about student knowledge as written proofs.

It should also be noted that WeBWorK, an open-source online homework system for math and science courses [216] supported by the Mathematical Association of America (MAA) and the US National Science Foundation (NSF), contains a Draggable Proof problem type that supports a JavaScript-enabled collection of drag and drop statements that can be auto-graded.

2.3 Definition and Characteristics

Parsons problems are tasks where the user needs to construct a solution by placing fragments into a correct order. In computer science education, the fragments are typically source code lines while the solution is a working computer program. The presentation of a Parsons problem to students includes a problem statement that specifies solution criteria, and a limited set of fragments that they then need to arrange into a correct order. Initially, fragments are typically presented in a separate area from a solution area. Students

[2] https://www.duolingo.com/

Characteristic	
Purpose and Use	Parsons problems are utilized in instruction and have a pedagogical purpose. They are often used as a part of a course. They can be used both to scaffold learning (more common) and for assessment (less common).
Problem Statement	Parsons problems feature a problem statement that is typically a description of a program that needs to be reconstructed. There is always an objective for the students, and reaching the objective is often verifiable. The problem statements are focused and have explicit goals (i.e. Parsons problems are not open-ended).
Atomicity	A limited number of fragments are provided. Each fragment typically represents a line but a fragment could contain more than one line. Although less common, fragments can also be elements of a line.
Problem Space	The problem space is constrained by limiting the number of available fragments and generally not allowing their reuse. However, Parsons problems can also feature distractor fragments that should not be used as well as possibilities for completing fragments (by e.g. providing values to variables or typing into part of the fragment).
Constructing a Solution	Parsons problems typically begin with an empty solution space into which the fragments are positioned in order. However, some Parsons problems only use only a single area with the objective of reordering the fragments within that area.
Correctness and Feedback	Parsons problems typically have a correct solution specified by clear criteria. They can often be automatically graded. Some systems require that the student solve the problem in a specified number of moves to be considered correct or in a set number of attempts. Parsons problems that are used for practice typically provide immediate feedback on the learner’s solution. There are two types of feedback: execution-based or line-based. Execution-based feedback is provided by executing the code, while line-based feedback is typically given by highlighting one or more fragments to indicate they are wrong or in the wrong place.
Modality and User Interface	Parsons problems are commonly worked on in an interactive drag-and-drop environment. This allows the use of them both within a browser and a mobile environment. Fragments in Parsons problems also often snap into place when moved, disallowing gaps between fragments. The fragments may feature locations into which text can be input (e.g. variable values). Some environments execute constructed programs in a text-based environment and show the output line by line, while other environments feature graphical output (e.g. Turtle graphics).
Syntax	When Parsons problems are related to a programming language, they commonly use the syntax from that programming language. A program represented using Parsons problems can be syntactically incorrect. The syntactically incorrect candidates solutions manifest primarily in two ways: (1) fragments that are ordered so that the resulting syntax is incorrect, and (2) distractors that are syntactically incorrect.
Scaffolding	Parsons problems are a type of scaffolding for learning programming. Parsons problems can include additional scaffolding as well. (e.g. In the Runestone ebook platform when students ask for help on a Parsons problem the system can remove a distractor or combine fragments.)
Fit and Expected Time on Task	From the instructional perspective, Parsons problems are intended to be helpful for student learning. This is reflected in the design of the problems. The presented problems should be appropriate to the current level of the learner; solving a problem typically requires from around a minute to less than 10 minutes depending on the number of fragments and complexity of the problem.

Table 1: Characteristics of Parsons Problems

then arrange the fragments into a correct order within the solution area.

In Table 1, we present a set of characteristics that are often present in Parsons problems. The characteristics can be used to describe, if not define, typical examples of Parsons problems and distinguish them from other approaches to programming, such as (1) Block-based programming, and (2) Text-based programming.

Block-based programming differs from Parsons problems in terms of the problem statement, the problem space, and the user interface. Typically block-based programming environments are open ended, while Parsons problems are not. Parsons problems typically also have a very limited set of available fragments, while Block-based programming environments provide a broader range of options to choose from. Finally, in Block-based programming, there are usually visual cues in the interface to indicate and constrain how certain blocks fit together (e.g. how a conditional statement could be constructed).

Text-based programming differs from Parsons problems in terms of atomicity, modality, and user interface, and expected time on task. In text-based programming, one writes code character-by-character (often supported by a programming environment), but there are no larger fragments. While Parsons problems are typically fast to complete, on the order of minutes (if not faster), completing a program using text-based programming in introductory programming courses may take hours or even days, depending on the complexity of the problem statement.

The characteristics in Table 1 can be used to describe the space of variations of Parsons problems. Parsons problems typically have a stated objective, namely the problem statement, and a limited problem space that facilitates construction towards that objective. However, variants of Parsons problems can be defined to further limit the problem space, thus providing additional scaffolding, or to expand the problem space, often requiring the learner to navigate through known difficulties.

For example, one highly-constrained variation of Parsons problems involves indentation, in which the lines of code are provided in the correct order and a solution is constructed by moving blocks horizontally to the correct level of indentation [175]. The problem space is expanded by requiring blocks to be placed in the correct order vertically as well as to be indented horizontally [109].

Another popular variation of Parsons problems increases the problem space by adding distractor blocks, which are extra fragments that are not needed in a correct solution. Distractors can be shown visually paired with each correct fragment [34, 43, 69] or randomly distributed among the correct fragments. This 'visual pairing' of correct and distractor blocks is generally recommended, as it can be overwhelming for students when a large number of distractors are randomly intermixed with correct options [43]. Parsons and Haden hypothesized that distractors would help learners learn to recognize common syntax and semantic errors. Distractors can also be used to make a Parsons problem more difficult and harder to game.

Faded Parsons problems [225] further expand the problem space by providing some incomplete blocks where the student needs to type missing code within the block to complete the solution. These problems are a variation of the atomicity characteristic, where blocks may have editable components. By limiting the amount of code-writing within selected blocks, faded Parsons problems provide one approach to scaffold a transition from block ordering to code writing. In an alternative approach to support code writing, students may start with a code-writing problem while giving them an option to fall back to a Parsons problem should they face difficulty in constructing the solution [106].

Design-level Parsons Problems are a variation of the syntax characteristic, where the blocks represent abstract steps at the design level rather than low-level source code or pseudocode. Such problems can be used to help students plan solutions at a high-level before they begin implementation with code, or simply to develop problem solving skills and gain familiarity with design strategies [82, 83].

The characteristics in Table 1 can also be used as the basis for formulating novel variations that could serve as the focus of future study. For example, varying both the problem space and the correctness and feedback characteristics, one could imagine style-based Parsons problems that feature multiple distractor fragments with poor code style (e.g. misleading variable names or indentation) where students are tasked with identifying and using the fragments that follow good style conventions.

2.4 Prior literature review on Parsons problems

Du, Luxton-Reilly, and Denny conducted a literature review on research related to Parsons problems in 2020 [48], which to the best of our knowledge is the only published peer-reviewed literature review on the topic. They focused on three research questions related to 1) motivations for the use of Parsons problems over other types of learning activities, 2) the different features and variations of Parsons problems that have appeared in the literature, and 3) how Parsons problems have been used in computing education.

Their first research question related to why Parsons problems are used, and they found four main motivations for using Parsons problems. First, they found that one claimed advantage of Parsons problems is to identify student difficulties. In traditional code writing exercises, it can be hard to separate student's syntactic and semantic issues. Other claimed benefits of Parsons problems are learning from immediate feedback, improving student engagement and reducing cognitive load.

There have been multiple variations of Parsons problems. In their review, they found that the most commonly used features that give rise to variations are related to scaffolding (e.g. whether code is correctly formatted or whether students have to format it), distractors (e.g. no distractors, paired distractors, jumbled distractors), and feedback (e.g. whether feedback is line-based or execution-based). Other rarer variations they identified include, for example, faded Parsons problems and Parsons problems where only part of the code needs to be organized.

Lastly, Du et al. studied how Parsons problems have been used in computing education. They identified two main use cases – paper-based exams and as a tool for student learning (e.g. using Parsons problems as regular practice exercises).

2.5 Related Theories

Experts have acquired extensive knowledge through thousands of hours of practice which affects what they notice and how they organize, represent, and interpret information in their environment [66, 204]. This, in turn, affects their ability to identify patterns and solve problems [89]. Thus, practice is crucial for learning, but it must be the right kind of practice [65]. Practice must include feedback and challenge the learner, but not be so difficult that it overwhelms the learner [27]. Learning is optimized when the learner is kept in Vygotsky's Zone of Proximal Development [219], which means that they are given problems that they can not solve independently but can solve with assistance. Writing code from scratch often takes an unpredictable amount of time and learners with no prior experience often need help and better feedback than compiler errors or incorrect results [19, 187]. Parsons problems can provide practice with immediate feedback and the difficulty of the problem can be adapted to keep the learner in the Zone of Proximal Development. Research on Parsons problems draws from theories and work on cognitive load, worked examples, self-efficacy, as well as metacognition and self-regulation.

2.5.1 Cognitive Load Theory. John Sweller developed cognitive load theory in the late 1980s and has continued work on the theory for decades [172, 173, 207, 211]. The theory describes three types of memory: sensory memory, working memory, and long-term memory. Learning occurs when new information is processed in working memory and then added to knowledge representations (schemas) in long-term memory [27]. Working memory has a limited capacity [159], and if that capacity is needed in its entirety to process new information, it cannot be used to modify or build schemas which is necessary for long-term retention of new information. Instructional materials can be designed to maximize the cognitive load dedicated to building schemas.

The amount of cognitive load a learner experiences is based on three components: the difficulty of the material or task, the way the instruction is designed, and strategies used for constructing knowledge. The difficulty of the material or task depends on the learner's prior knowledge and the complexity of the task [49]. Parsons problems, as a type of code completion problem, should have a lower cognitive load than a problem that requires the learner to write the code from scratch, because they constrain the problem space [218]. Moreover, Parsons problems transform the task of creating programs by removing the necessity of remembering the programming language syntax, which further decreases cognitive load[3].

2.5.2 Worked Examples. One of the original goals for Parsons problems was to expose students to an expert's solution to a problem [177], which is also called a worked example [36]. The worked example effect, in which learning is improved by studying worked examples versus solving problems, is one of the most well known effects predicted by Cognitive Load Theory [8, 208]. Research has been conducted on worked examples in math [10, 37, 210, 214, 236], physics [192, 194, 223] and computer programming [166, 179, 235]. Worked examples are particularly useful for initial cognitive skill development, such as in learning to program [191]. However, students do not always learn from worked examples [53], as learning requires cognitive effort. The worked example effect decreases and can even reverse as expertise increases [209]. This is called the expertise reversal effect. Students learn best when worked examples are interleaved with practice problems that are similar to the worked examples [215]. Another argument in favor of worked examples is that students prefer learning by studying examples versus learning by reading text [137]. Parsons problems have been used as a type of interleaved practice after worked examples [60, 104, 105].

2.5.3 Self-Efficacy. Self-efficacy is the belief that you can succeed in a specific situation or accomplish a task [9]. People eliminate possible vocations from consideration if they do not believe that they can succeed in those fields [9]. Students who encounter errors while programming experience negative emotions that impact their self-efficacy [124]. High self-efficacy improves persistence in a field, while low self-efficacy increases the odds that students will fail or change majors [52]. Negative experiences in courses tend to affect women more than men [51, 154] which may be one reason that women are more likely to leave majors than men, even if they have better grades than the men who stay [120]. Students from underrepresented groups tend to have less prior experience in computing [21, 153, 154, 221], which makes them more at risk for failure. One argument for using Parsons problems is to try to improve student success on early programming tasks in order to increase their self-efficacy, which could increase the diversity of computing students in general.

2.5.4 Metacognition and Self-Regulation. Another motivation for using Parsons problems could be for scaffolding novice programmer metacognition [186]. Metacognition is, simply put, thinking about thinking. Self-regulation is a metacognitive skill that refers to a learner's ability to reflect on their own learning process, understand it, and change it if necessary. Setting goals, motivation, process inspection, and evaluation are all key concepts. One common argument about learning programming is that it is so difficult to master the cognitive skills (learning new syntax, thinking computationally, etc.) that metacognitive skills are often underdeveloped or not present in the domain [140, 182, 184]. Although multiple recent attempts have been made to increase metacognition with novice programmers [45, 185], only a few papers focus on the effect of Parsons problems on novice programmer metacognition [82, 83, 183].

3 SYSTEMATIC LITERATURE REVIEW

Reviews of the research literature in computing education are common. An excellent summary of reviews conducted over the past 20 years that categorize or evaluate computing education literature was recently provided by Heckman *et al.* [99]. A systematic literature review is one type of review method that follows a protocol in which the search process is clearly documented so that its rigor and completeness can be assessed. Although there are various guidelines for conducting discipline-specific systematic literature reviews, it is common for computing education researchers to adopt (and often adapt) the process defined by Kitchenham and Charters for software engineering researchers [125]. The primary reasons they cite for conducting systematic reviews are to provide a complete background on a topic to position new research activities, to identify gaps in current research, and to summarize existing empirical evidence for the benefits and limitations of some approach. All of these reasons align with the goals of our work, and our literature review is guided by their protocol.

At a high level, the three phases of the review process are planning, conducting, and reporting [125]. The planning phase involves

[3] If the goal of instruction is to teach how to compose a working solution, learning and remembering the programming language syntax can be considered extraneous to learning. Few papers make an explicit distinction between the types of cognitive load – intrinsic or extraneous – being measured or predicted [50].

defining research questions and developing a review protocol. The conducting phase involves identifying research and extracting and synthesizing data from selected studies. The reporting phase is primarily focused on organizing the results for publication. We define our research questions in Section 3.1 and our process for identifying literature in Section 3.2. In subsequent sections, we describe our selection processes (title and abstract scanning, and inclusion and exclusion criteria) and our data extraction protocol. We present the main findings of the review in Section 3.7

3.1 Research Questions

The goal of our review is to produce an up-to-date survey of the literature on the use of Parsons problems in computing education research. We state our overarching research question as follows:

- How have Parsons problems been investigated in the computing education research literature?

We are particularly interested in documenting the range of contexts in which Parsons problems have been studied and the various tools that have been developed to support the delivery of Parsons problems to students. In particular, new tools continue to emerge that support novel features of Parsons problems and we wish to understand this variety. We also wish to explore the various motivations and interests of the computing education research community related to the use of Parsons problems. Finally, we wish to catalogue the existing evidence to date for the benefits that Parsons problems might offer to learners. We define the following three research questions that we use to guide our literature review and the extraction of data from primary studies:

- RQ1: In what contexts and with what types of features have Parsons problems been studied?
- RQ2: What research questions have driven the existing literature on Parsons problems?
- RQ3: What evidence exists for the claimed benefits that Parsons problems offer, and for their limitations, and what is the quality of that evidence?

3.2 Identification of Relevant Literature

To identify relevant literature, we conducted database searches using three sources: (1) ACM Digital Library (Guide to Computing Literature); (2) IEEE Xplore; and (3) Scopus. These three databases are commonly used in reviews of the computing education literature and there is evidence that they yield a greater proportion of relevant results in this domain than other databases [144]. As described later in Section 3.4, we combine these searches with a forward snowballing phase (examining all articles that cite the seminal Parsons and Haden paper [177]), to achieve a broad coverage of the literature.

Kitchenham and Charters suggest several strategies for refining search terms, including comparing the results of a given set of search terms with lists of known primary studies and experimenting with various combinations of search terms [125]. The first strategy helps assess the completeness of the search, as known primary studies should be included in the set of results if the search terms are well chosen. The second strategy helps reduce the number of articles returned that are not relevant and would need to be filtered in an exclusion step. For each strategy, we now briefly describe the approach we used for refining our search terms.

3.2.1 Validation set from existing review.

To our knowledge, the only published peer-reviewed literature review related to Parsons problems is "A Review of Research on Parsons Problems" published in 2020 by Du, Luxton-Reilly, and Denny [48] (see Section 2.4). Our review brings this work up to date, widens the range of venues searched, and increases the scope by including a focus on the empirical evidence for the effectiveness of Parsons problems.

This prior work serves as a useful validation tool for our search. The authors identified 34 primary studies from an initial set of 325 search results, however only a subset of these papers were presented and identified as primary articles. In particular, 13 papers were listed in the results as characterizing the different varieties of Parsons problems and the contexts in which studies were conducted. We use these 13 papers as a validation set for our search.

3.2.2 Exploring keywords.

We explored around a dozen search string variations, all of which included some combination of the terms "parsons" and either "problems" or "puzzles". We considered several other terms that we felt had the potential to yield relevant results, such as "code puzzles". All such candidate terms were evaluated systematically by running a search with and without the term, and manually reviewing differences in the results. As an example, including the term "code puzzles" resulted in 13 additional papers when searching the ACM Digital Library, however none of these 13 papers were relevant to our topic and so we excluded the search term.

Ultimately, we observed that the terms "parson" or "parson's" or "parsons", combined with "problems" or "puzzles" or "programming", produced a good outcome. These terms returned 12 of the 13 papers in our validation set, missing only one paper which was published in a venue not indexed by our databases (the Journal of Information Technology Education). We resolve this omission through a snowballing phase, outlined in Section 3.4.

We defined variations of the term "parsons" manually, as shown above, rather than relying on the use of a wildcard character. This follows guidance provided by Loksa et al. regarding keyword variations in systematic literature reviews, as the behaviour of wildcard matches is not always intuitive [140]. Queries were refined for each database appropriate to their functionality (a commonly required step when searching digital databases [28]). To provide an example, the final query used for the ACM Digital Library was as follows:

```
[All: "parson problems"] OR
[All: "parson puzzles"] OR
[All: "parson programming"] OR
[All: "parson's problems"] OR
[All: "parson's puzzles"] OR
[All: "parson's programming"] OR
[All: "parsons problems"] OR
[All: "parsons puzzles"] OR
[All: "parsons programming"]
```

In general, it can be challenging to formulate good search strings for literature reviews, especially when there is a lack of standard terminology and thus broad terms must be used which can result in many irrelevant matches [32, 227]. In our specific case, we have the advantage that the term “Parsons” is a commonly accepted term for the subject of our review, and would typically appear in the full text or reference list of a paper (both of which are indexed in a full text search using the ACM Digital Library). This is a point that was also acknowledged by Du, Luxton-Reilly, and Denny in their review on Parsons problems [48].

Our final search was conducted on the 10th of May, 2022. Table 2 summarizes the searched databases and the number of matching results from each. After merging the three result sets, we removed duplicates based on title and DOI. This led to the removal of 152 articles, leaving a total of 677 articles for further analysis.

Database	Results
ACM Digital Library (Guide to Computing Literature)	197
IEEE Xplore	247
Scopus	385
Total results	*829*
Total results (duplicates removed)	*677*

Table 2: Database search results.

3.3 Title and Abstract Scanning

Identification of relevant articles was followed by a title and abstract scanning phase, during which the researchers read the title and abstract of each paper to identify articles that were clearly out of scope. For each article, the potential exclusion was determined by two researchers – if both researchers agreed on the exclusion of an article, the article was removed. If either of the two researchers considered that an article *might* be relevant, it was included for further analysis.

We calculated a Cohen’s kappa coefficient between the raters, which suggested an almost perfect agreement (kappa = 0.973). However, it should be noted that the high value is partly due to the fact that in our process the second rater was not required to rate articles that the first rater had marked as potentially being relevant, as such articles were automatically included for further consideration after the scanning phase. Disagreements occurred only when the second rater of an article wished to consider it for inclusion after the first rater had excluded it.

This approach follows the suggestion of Brereton et al. for erring on the side of caution when filtering primary studies [28]. The title and abstract scanning phase led to the removal of further 291 articles, leaving a total of 386 articles for application of our full inclusion and exclusion criteria.

3.4 Forwarding snowballing

Snowballing is an alternative strategy to database searching for identifying relevant literature and is based on citation information [75]. We employed a forward snowballing strategy which involved finding articles that cited existing work that was known to be relevant. In our case, we used the original paper by Parsons and Haden [177] as the seed paper and found all articles that have subsequently cited it using the ‘Cited by’ feature of Google scholar. The goal of this snowballing phase was to capture other relevant literature that may not have been indexed by the databases we used in our database search. In particular, this included the one paper from our validation set that was not returned by our database search (see Section 3.2.1) but which did cite the Parsons and Haden paper.

A total of 306 papers were found by this snowballing step. We initially removed 167 duplicate papers that already appeared in the result set from our database search, and then performed a title and abstract scan for relevance on the remaining papers. This resulted in a total of 108 papers for manual review. Figure 2 provides a high-level overview of the sources for our primary studies, the number of papers removed as duplicates and during title and abstract scanning and the final result set after manual application of our inclusion and exclusion criteria.

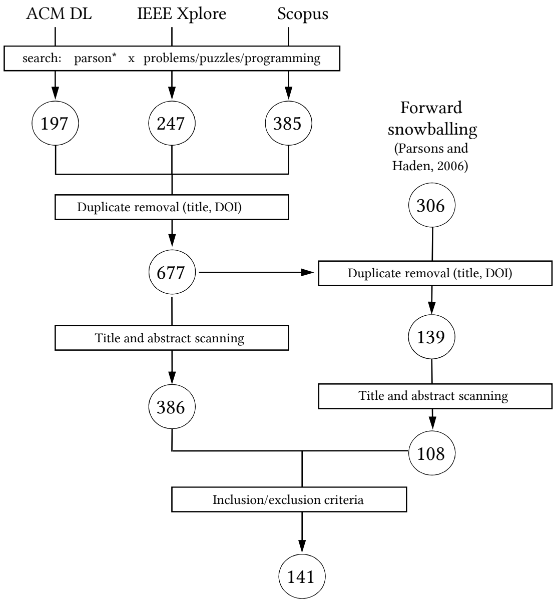

ACM DL
IEEE Xplore
Scopus
search: parson* x problems/puzzles/programming
197
247
385
Forward snowballing (Parsons and Haden, 2006)
Duplicate removal (title, DOI)
306
677
Duplicate removal (title, DOI)
Title and abstract scanning
139
386
Title and abstract scanning
108
Inclusion/exclusion criteria
141

Figure 2: An overview of the search and snowball process illustrating the number of papers resulting from each step.

3.5 Inclusion and Exclusion Criteria

We manually examined 494 articles that passed the title and abstract scanning phase. For each of these articles, we determined whether or not it was relevant to our literature review, and if so, we extracted data from the paper through a structured form. Our overarching research question was to explore how Parsons problems have been investigated in the computing education literature, and so our key inclusion criteria was that relevant articles *must discuss Parsons problems in a computing education context.* This required a definition for Parsons problems relevant to computing education, which we constructed as follows (discussed more broadly in Section 2.3).

> *Parsons problems are a type of exercise used for learning and assessing the ability to construct programs (‘programs’ = intentionally allowing broad interpretation, e.g. ordering higher level plans, while limiting the scope*

> *to a programming context). Parsons problems feature a limited set of fragments that the user needs to use to produce a solution ('limited' = not open-ended, 'use' = intentionally allowing variations where the fragments are in a single area). There can be extra fragments that are not needed for a solution, and the solution may require some text input e.g. entering variable values into the fragments.*

To determine whether or not an article was relevant to our review, we created three inclusion criteria and included papers that fulfilled any one of these. The inclusion criteria were as follows:

IC1 Contains empirical results on the use of Parsons problems and/or collects data from the use of Parsons problems
IC2 Describes a system/tool for presenting/delivering Parsons problems
IC3 Describes the use of Parsons problems for teaching

We now give brief examples to illustrate these criteria. An exemplar article that would fit IC1 is "Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code" by Haynes and Ericson [98]. In that article, the authors studied the difference between completing adaptive Parsons problems and writing the equivalent code. This article matches IC1 as the authors collected empirical data related to the use of Parsons problems.

For IC2, a good exemplar paper that matches the criteria is "A Mobile Learning Application for Parsons Problems with Automatic Feedback" by Karavirta et al. [119] where the authors presented a mobile application for practicing Parsons problems called MobileParsons as well as outlined improvements to the open source Parsons problem library js-parsons [109]. This is a clear match for IC2 as the article clearly describes the MobileParsons tool.

Finally, one exemplar that matches IC3 is the paper "Tasks That Can Improve Novices' Program Comprehension" by Shargabi et al. [199]. In this work, instructors from 13 universities were asked to rank a list of 14 commonly used instructional tasks for their perceived effectiveness. Parsons problems were one of the task types listed on the survey, and were rated as moderately effective as a teaching activity for program comprehension, but this paper did not present any empirical data on the use of Parsons problems or describe a tool for their delivery.

When considering works to exclude, we omitted articles that were not written in English (EC1) as we could not easily extract data from such papers, and articles that were very short (EC2) as these typically represent posters and abstracts and are generally less rigorous or represent work in progress. As a quality control measure we also required that articles be peer-reviewed (EC3), which also meant that theses and dissertations were also excluded (EC4). However, we acknowledge that it is common for articles from monographs to be published in peer-reviewed venues in which case we would still capture them. Finally, we excluded any paper that was not relevant to our goal of understanding how Parsons problems have been investigated in the computing education literature. We defined two separate exclusion criteria to differentiate between papers that were completely unrelated to the topic of Parsons problems (EC6), and those which did make fleeting mention of Parsons problems, but where the Parsons problems were not related to the research questions or goals of the paper, and where there was no relevant discussion of approaches or findings in the methods or results (EC5).

To illustrate, one typical example of such a paper in the category EC5 is the work by Oyelere et al., published in the Journal of Education and Information Technologies, which described the design and development of a mobile learning application called MobileEdu for teaching computing in the Nigerian higher education context [171]. In this paper, the description of MobileEdu does not include any use of Parsons problems, however the paper presents a review of mobile learning applications that have been previously used in computing education and cites work on the MobileParsons tool which is a relevant work to our review [107].

The six exclusion criteria were defined as follows: (Fulfilling any one of these criteria was sufficient for exclusion.)

EC1 Article is not written in English
EC2 Article length is less than or equal to 2 pages
EC3 Article is not peer-reviewed
EC4 Article is a thesis or a dissertation
EC5 Parsons problems are not related to the research questions/goals of the paper, and there is no relevant discussion in the methods or results
EC6 Not related to Parsons problems

3.6 Data Extraction

For all papers that were selected for inclusion, we used a data extraction process to extract and record information in a consistent format. An initial data extraction form was defined and published on Google forms. As recommended by the guidelines for this phase that were suggested by Brereton et al. [28], this form was refined through a process of iteration that actively involved seven members of the research group performing the review. This refinement process included a training phase in which all researchers extracted data using the form for five practice articles (two of which were reviewed by all researchers, and three which were reviewed by pairs of researchers). This training phase involved regular discussions with the whole group and led to refinements such as the inclusion of definitions for each field directly on the form.

In total, 25 fields were included on the data extraction form. In addition to this main set of items, a brief four-item quality assessment questionnaire was created to assess the quality of the state of research into Parsons problems. We adapted these quality assessment items from the more comprehensive list used by Ihantola et al. [110]. Once the extraction sheet was finalized, the researchers continued to meet weekly, over a period of approximately three months, to track regular progress for the data extraction and to resolve any issues; whenever a member of the team had concerns or questions about an article or its extraction, it was marked for discussion at one of the weekly meetings where it was discussed with the group and resolved. The final extraction sheet, including the quality assessment questionnaire, is outlined in Appendix A. We have made our complete literature review dataset available as an Open Science Framework (OSF) repository: https://bit.ly/3AyQLhv.

3.7 Analyses

3.7.1 Contexts and Features. To answer Research Question RQ1, *In what contexts and with what types of features have Parsons problems been studied?*, we evaluated the quantitative questions in our extraction sheet (see Appendix A). For questions with predefined fields, we computed frequencies of occurrence for each category in the extraction sheet. For open-response answers (e.g. “What concepts are being taught with Parsons problems?, or “other” in some questions,) we first cleaned the answers and then standardized before computing frequencies for each category. We include in our report categories with at least two occurrences.

Bibliometric data, such as year and venue of publication, were extracted directly from the BibTeX entries that were recorded for each paper.

3.7.2 Explored Research Questions. To answer Research Question RQ2, *What research questions have driven the existing literature on Parsons problems?*, two researchers worked together side-by-side to identify research themes from all of the included Parsons problems literature, identifying all key themes which occurred in more than a single article. For articles that specifically identified their research questions or hypotheses, these were used directly in the thematic analysis. For articles that did not directly list research questions or hypotheses, research themes were identified from the key contributions as discussed in the abstract, discussion, and summary sections of the respective articles.

3.7.3 Evidence. To answer Research Question RQ3, *What evidence exists for the claimed benefits that Parsons problems offer, and for their limitations, and what is the quality of that evidence?*, we examined every paper that had been tagged with IC1 during our systematic literature review. This tag had the description ‘*Contains empirical results on the use of Parsons problems and/or collects data from the use of Parsons problems.* Specifically, we looked at what type of evidence the authors of these papers were attempting to present and then examined the quality of that evidence. Many papers were tagged with more than one type of evidence tag. After grouping papers by evidence type (where a paper can appear in more than one list), we examined the following characteristics about those papers based on data collected during the systematic literature review[4]:

- Which (programming) languages were used?
- What was the size (n) of the study?
- How many Parsons problems were used in the study?
- What was the type of study (qualitative/quantitative)?
- Were there clear research questions?
- Was the methodology clear?
- Was there sufficient clarity to reproduce the study?
- Were there two or more groups? (i.e. at least one control and one experimental)
- Did the study provide a clear measurement on the efficacy of Parsons problems?
- How was the study delivered? (in-person, online, hybrid, other)

4 RESULTS OF THE LITERATURE REVIEW

4.1 Overview of Included Articles

As illustrated in Figure 2, a total of 386 papers passed the title and abstract scanning step for the database search, and 108 papers passed this step for the snowballed papers. Following manual examination of these papers, 141 matched at least one of the inclusion criteria (IC1 – IC3) and the remaining 353 were excluded. The full result set of 141 included articles is provided in Appendix C. For each paper examined, we recorded all matching inclusion criteria and the first matching exclusion criteria. Table 3 provides a breakdown of how the included and excluded papers were distributed across these criteria.

The most frequently matched exclusion criteria was EC5, *Parsons problems are not related to the research questions/goals of the paper, and there is no relevant discussion in the methods or results.* Papers that matched this criteria did include some mention of Parsons problems but they were not the focus of the paper. As illustrated in an earlier example, one of the common reasons for such classifications was papers citing and briefly discussing one or more relevant articles on Parsons problems as part of their literature review or background work, but with no further treatment.

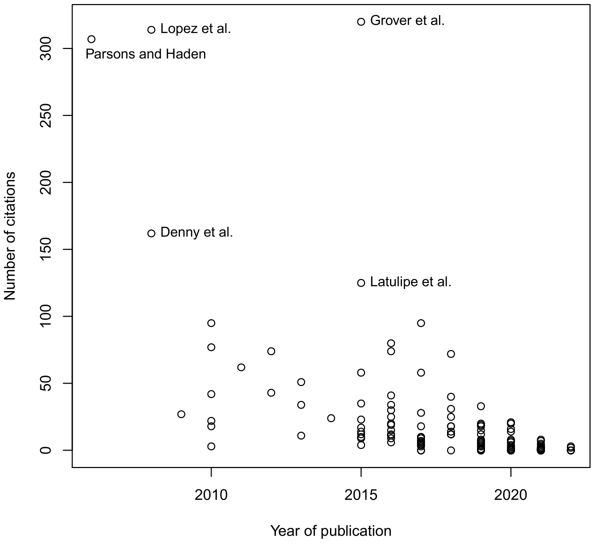


Figure 3: Number of citations plotted against year of publication for included articles (publications with more than 100 citations annotated). Citations counts per Google Scholar (9th July, 2022)

4.1.1 Publication trends and venues. Table 4 lists the twelve most cited publications[5] that refer to Parsons problems. Interestingly, only five of the articles [43, 62, 101, 165, 177] are specifically research on Parsons problems (i.e. they mention Parsons problems in the title), and seven others matched our inclusion criteria while not solely focusing on exploring Parsons problems. (For example, they might have used Parsons problems in teaching alongside other

[4] These characteristics were also reported for all papers in RQ1

[5] According to Google Scholar.

IC/EC	Inclusion criteria							Exclusion criteria					
	1	2	3	12	13	23	123	1	2	3	4	5	6
# of matches	33	20	21	37	14	6	10	5	45	11	28	204	45
	Total: 141							Total: 338					

Table 3: Inclusion and exclusion criteria matched by articles found in the literature review.

Year	Author	Title	Venue	Citations
2015	Grover, Pea and Cooper	Designing for deeper learning in a blended computer science course for middle school students	Computer Science Education	320
2008	Lopez, Whalley, Robbins and Lister	Relationships between reading, tracing and writing skills in introductory programming	ICER	314
2006	Parsons and Haden	Parson's programming puzzles: A fun and effective learning tool for first programming courses	ACE	307
2008	Denny, Luxton-Reilly and Simon	Evaluating a new exam question: Parsons problems	ACE	162
2015	Latulipe, Long and Seminario	Structuring flipped classes with lightweight teams and gamification	SIGCSE TS	125
2010	Schulte, Clear, Taherkhani, Busjahn and Paterson	An introduction to program comprehension for computer science educators	ITiCSE WG Reports	95
2017	Ericson, Margulieux and Rick	Solving Parsons problems versus fixing and writing code	Koli Calling	95
2016	Morrison, Margulieux, Ericson and Guzdial	Subgoals help students solve Parsons problems	SIGCSE TS	80
2010	Lister, Clear, Simon, Bouvier, Carter, Eckerdal, Jacková, Lopez, McCartney, Robbins, Seppälä and Thompson	Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer	ITiCSE WG Reports	77
2012	Helminen, Ihantola, Karavirta and Malmi	How do students solve Parsons programming problems?: an analysis of interaction traces	ICER	74
2016	Oyelere, Suhonen and Sutinen	M-Learning: A new paradigm of learning ICT in Nigeria	Int. J. of Interactive Mobile Tech.	74
2018	Brown and Wilson	Ten quick tips for teaching programming	PLoS computational biology	72

Table 4: The 12 most cited publications (according to Google Scholar) that refer to Parsons problems

activities.) The twelve most cited papers included in the literature review have been published in a variety of venues – interestingly, the Australasian Computing Education conference has two highly cited publications, attesting to the region of the origin of Parsons problems.

The complete set of papers included in our literature review were published in a very diverse set of venues. The majority of those papers were published in conferences (75.9%) or journals (19.2%). Workshops (5.0%) published the remainder of the papers. The ACM Conference on International Computing Education Research (ICER) was the leading publication venue with 9.9% of the included papers, closely followed by the ACM SIGCSE Technical Symposium (SIGCSE TS – 9.2%), the ACM conference on Innovation and Technology in Computer Science Education (ITiCSE – 7.1%), the Koli Calling International Conference on Computing Education Research (6.4%) and the ACM CHI Conference on Human Factors in Computing Systems (5.7%). Computer Science Education (CSE) was the leading journal publication venue (4.2%), with ACM Transactions on Computing Education (TOCE) and the Journal of Computing Science in Colleges (JoCSiC) contributing with two papers each (see Table 5). The remainder of our dataset (35.21%) was spread over a number of publications with one included paper each, a testimony to the impact of Parsons problems and their widespread usage.

4.1.2 *Citation trends.*

Figure 3 plots the year of publication for every paper in our data set against the number of citations that the paper has attracted. We compute the number of citations using Google Scholar, and the figures were accurate as of 10th July, 2022. As would be expected, this chart exhibits a general negative correlation between publication year and number of citations, given that more recent papers have less time in which to be visible and

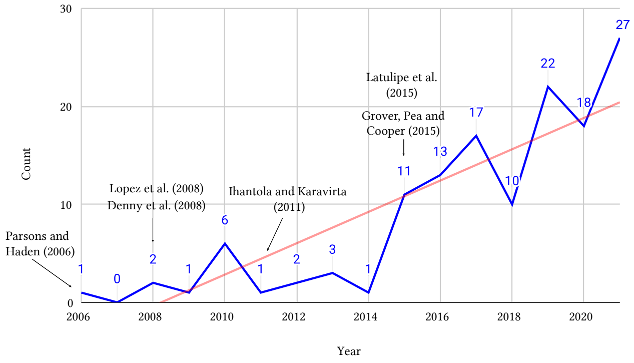


Figure 4: The number of publications per year (2006 – 2021). There were 6 publications in 2022 at the time of writing, but they are not shown on this figure due to the timing of our search.

Venue	Papers
ICER	14
SIGCSE TS	13
ITiCSE	10
Koli Calling	9
CHI	8
CSE	6
Australasian Computing Education Conference (ACE)	6
American Society of Engineering Education Conference	5
International Conference on Educational Data Mining	4
IEEE Symposium on Visual Languages and Human-Centric Computing	4
International Conference on Computers in Education	4
JoCSiC	2
Workshop in Primary and Secondary Computing Education	2
IEEE Frontiers in Education	2
ACM TOCE	2
Other venues	50

Table 5: Number of included papers by venue.

cited. Annotated on the figure are the five most cited papers (further details of these papers appear in the first five rows of Table 4). In chronological order, the first of these is the original paper by Parsons and Haden [177]. This was followed in early 2008 by Denny et al. which described the first use of Parsons problems as an exam question and presented several novel variants of such problems [43]. Later the same year, Lopez et al. also presented data from the use of Parsons problems on a pen-and-paper exam, which they used to explore the relationship between the skills of code comprehension and code writing [142]. Seven years later, early in 2015, Latulipe et al. explored the use of in-class teamwork to make learning more social and effective, making use of Parsons problems as hands-on activities, although they did not provide empirical data on use of these [136]. Again, later that same year, Grover et al. described the design of an introductory computer science course for middle school students where the quiz questions provided to students included Parsons problems in the form of Scratch blocks that were jumbled [91].

Figure 4 plots the number of papers in our data set that were published each year (Note that the year 2022 is excluded from the plot because the 2022 data was only partial at the time of this article). Highly cited papers prior to 2022 are indicated. The plot shows a marked upward trend in the number of publications on Parsons problems with a notable spike occurring after 2014. This spike may be accounted for by the emergence of online ebooks with Parsons problems. A review of the citations from the articles published in 2015 reveals references to the js-parsons implementation [108] and studies that use it (e.g. [101]).

In general, the emergence of Parsons problems and their use in the computing education research literature parallels the broader adoption and use of interactive online learning materials. In 2013, an ITiCSE working group explored the use and creation of interactive computer science ebooks [126], and an ITiCSE 2014 working group focused on interoperatibility of learning content, including increasing adoption of tools for teaching computing [30]. Both working groups discussed Parsons problems as examples of content that computer science ebooks could and do feature. The 2013 working group considered the possibility of having Parsons problems (from the technical viewpoint) either as independent components or as a part of other types of components that are included in the materials [126], while the 2014 working group referred to Parsons problems as a type of problem-solving support tool and more broadly discussed the possibilities of creating online learning materials that could include resources from various other learning platforms and resource repositories [30]. These working groups influenced and have been influenced by the developments of open online ebooks and ebook platforms for teaching computing, including the OpenDSA project [197] and the Runestone Academy project [158].

4.1.3 Quality assessment. Figure 5 shows the classification distribution for the four quality assessment items. We adapted these four items from the larger set of 15 items used in the review of educational data mining and learning analytics literature by Ihantola et al. [110], and our results mirror the results of this prior study, where the clarity of findings and the analysis of threats to validity were the most and least met criteria respectively. In our data set, the most frequently achieved quality criteria was "Are the results presented with sufficient detail?" and the least achieved criteria was the explicit presence of "threats to validity / limitations". Our results suggest that authors frequently do not consider and communicate limitations of their work or alternative explanations for their findings.

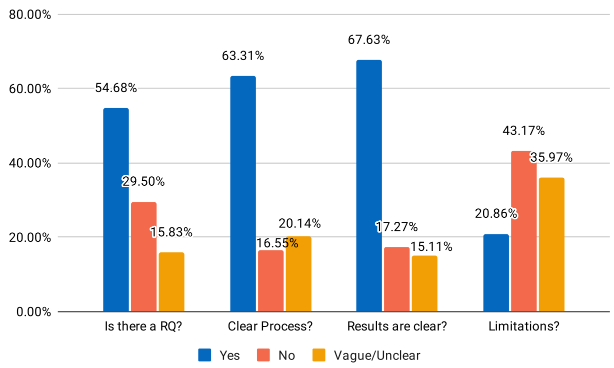


Figure 5: Quality assessment of included papers

4.2 RQ1: In what contexts and with what types of features have Parsons problems been studied?

4.2.1 Study contexts. We investigated the types of papers related to Parsons problems by classifying them as *lab-based*, i.e. a study conducted in a controlled environment such as a research lab; *classroom-based*, i.e. a study conducted in "the wild," such as in a classroom; a *system paper*, which describes a system used to present Parsons problems or a system that uses Parsons data with an educational purpose; or an *experience report*, where the author reflects on their personal experience, but does not involve collection and analysis of data from students or the system. Papers that do not fit in any of those categories were classified as *other*. Some papers (e.g., papers presenting a tool to deliver Parsons problems and which also evaluated its use and students' performance solving tasks) could be classified in multiple categories.

We found a larger number of classroom-based studies (36.4%) compared to lab-based studies (25.3%). There was also a fair number of system papers (22.8%), but slightly under half of the system papers (8.6%) did not include an evaluation or report on the usage of the system. Experience reports were less well represented in our dataset (9.9%), and only 5.6% of the papers could not be classified into any of the categories we included in the extraction form.

Not surprisingly, the vast majority (52.9%) of the included papers conducted studies at the tertiary level in introductory programming courses (CS1, CS2). Usage of Parsons problems at the tertiary level in other courses (e.g., data structures, databases) was far less prevalent (9.8%). Studies at primary levels (9.2%) and secondary levels (9.8%) are represented in similar numbers in our review. Life-long learning contexts (e.g., MOOCs) represent 4.6% of our sample. We could not ascertain the educational context in 9.2% of our papers (i.e., it was not reported clearly), and 4.6% of the studies were not associated with a specific educational level (e.g., they had no real world evaluation). Only a few papers (9.2%) conducted studies at multiple levels (e.g., both secondary and tertiary).

To understand the geographic distribution of this research, we recorded the country in which the study was carried out. Figure 8 overlays this frequency data on a map of the world, illustrating that the vast majority of studies have been conducted in the United States. The remaining studies were distributed throughout Canada, Australasia and parts of Asia and Europe. If a study sourced data from multiple countries, it was counted once for all countries where data was gathered. Most studies were conducted in a campus-based setting (36.2%). Blended or hybrid settings were used in 7.1% of studies and 15.6% of the studies were conducted in an online setting. Studies did not have a delivery setting in 25.5% of the papers in our review, and in 15.6% of the papers it was unclear which delivery setting was used.

We investigated if students received any marks/grades/credit for participating in the Parsons activities described in the papers. In 34% of the studies, students were graded when solving Parsons problems and in 9.2% of the papers students were not graded. We were unable to ascertain whether or not Parsons problems were graded in 31.9% of the papers. In 24.8% of the papers there was no intention to evaluate students' performance when solving Parsons problems.

In the majority of the studies (44.0%), Parsons problems were part of regular instruction, were presented as a bonus activity (8.5%) or as an extracurricular activity (9.9%). In 10.6% of the papers it was unclear how Parsons problems were inserted in learning activities, and in 27.0% of the studies they were not part of the instruction.

We investigated how extensive Parsons problems usage was in the studies in our review. In 27.7% of the studies, five or more problems were used, 8.5% of the studies used between two and four problems, and 9.2% used only one Parson problem. It was unclear how many problems were used in 34.8% of the studies and in 19.9% of the studies there was no description of Parsons problems usage.

4.2.2 Study participants. We looked into the size of each study by extracting the numbers of study participants. This data was extracted either from the contextual description or from the study description, depending on which numbers were available. The participant counts were then categorized into bins of increasing sizes starting from `1-20` and ending at `1001 or more`. Figure 6 summarizes the study participant numbers in the included articles.

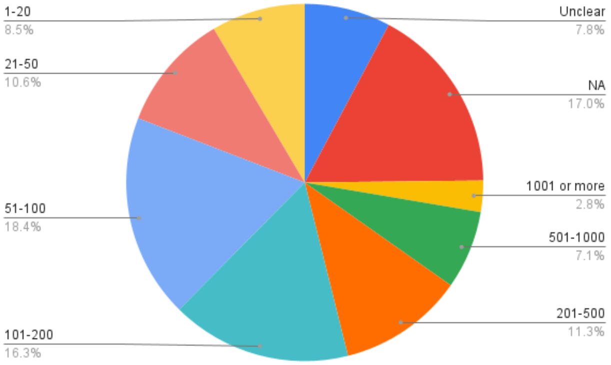


Figure 6: Study participants in included articles

Out of the 141 articles, 35 either had an unclear number of participants (i.e. not reported) or the number of participants was not applicable (not relevant due to e.g. the article being a system paper). Across the bins, the bin with the most papers was the 51 to 100 participants bin, with 26 classified articles. The bin with the least number of papers was the 1001 or more participants bin, with just 4 articles. Out of the articles with reported participant numbers, there were on average 455 participants, while the median number of participants was 102 – highlighting the fact that only a small number of studies had a very large number of participants.

4.2.3 Programming languages and concepts. Table 6 shows the distribution of different programming languages that are reported in the papers in our data set. Unsurprisingly, the three most common languages are Python, C/C++ and Java, which are the most commonly taught introductory languages. Much less common were papers that reported the use of Parsons problems with pseudocode, representing around 3.6% of the complete list of languages across the data set (some papers described tools that supported more than one language). One such example is the work by Malik et al., who present a tool to support their PAAM (Problem Analysis Algorithmic Model) teaching approach, which aims to improve both comprehension of problem statement requirements and problem solving skills [150]. For the problem-solving aspect, their tool includes a Parsons-like interface where students solve pseudocode problems. Students who used the PAAM approach throughout the semester showed improved perceptions around the understanding of problem statements and problem-solving strategies.

Language	#	Percentage
Python	49	29.17%
Java	32	19.05%
C/C++/C#	30	17.86%
Not PL Focused	19	11.31%
Other	14	8.33%
Unclear	7	4.17%
Pseudocode	6	3.57%
Looking Glass	5	2.98%
Scratch	3	1.79%
MATLAB	2	1.19%
JavaScript	1	0.60%

Table 6: Programming languages used

Identifying which concepts were used with Parsons problem was difficult to ascertain in our review. A significant number of works were unclear (30%) on what concepts were covered by Parsons problems related activities. – see Table 7. When concepts were explicitly presented, it was usual to find a list of several concepts that were covered by Parsons problems, and the majority of the papers (21.7%) stated learning loops as an instructional goal. Conditionals (10.6%), variables (6.7%), lists and arrays (8.2%), and functions (5.8%), typical introductory programming concepts, were also mentioned by the authors as instructional goals. More advanced topics such as recursion (1.0%), and objects and classes (3.4%) were less prominent in our review. Interestingly, concepts such as expressions, sequential commands, and I/O, also typically part of CS1 courses, were less often referred to as part of instructional goals (see Table 7).

4.2.4 Features of Parsons problems. In our review, we aimed to investigate how different features described in the literature were used. Particularly, we looked for features such as draggable code lines with parts that need to be filled in (Faded Parsons) [225], adaptive Parsons problems [59], usage of distractors (i.e. code lines that are not needed in the final solution) [95], optional fragments of code (i.e. similar code lines where you have to pick one or more but not all from a set) [62]. We assumed that papers could have used a combination of any of these features in their studies with the basic (i.e. draggable code lines) type of Parsons problems.

Concept	#	Percentage
Unclear	61	29.61%
Loops	45	21.84%
Conditionals	22	10.68%
Lists and Arrays	17	8.25%
Variables	14	6.80%
Functions	12	5.83%
Strings	8	3.88%
I/O	8	3.88%
Objects	7	3.40%
Sequential commands	6	2.91%
Expressions	4	1.94%
Recursion	2	0.97%

Table 7: Concepts targeted by Parsons problems

Where the authors did not explicitly state which kind of features were used, we attempted to extrapolate potentially used features from images in the paper. Many authors were, however, unclear on the kind of features present in Parsons problems related activities, so we could not ascertain which features were used in 34.6% of the papers. Basic features were present in the majority of the papers in which we could determine types of features used – 34.1%. Distractors, another feature that can be "easily" added to Parsons problems were used by 18.8%. As presented by Figure 7, features that require specific implementations such as adaptive Parsons were less common in our review.

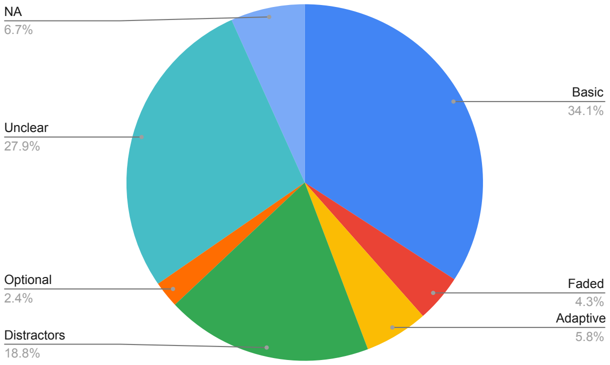


Figure 7: Features used in Parsons problems

4.2.5 Tools for Parsons Problems. Parsons problems have been delivered in a variety of modes and through a wide range of learning platforms. This includes as pen and paper activities, stand-alone

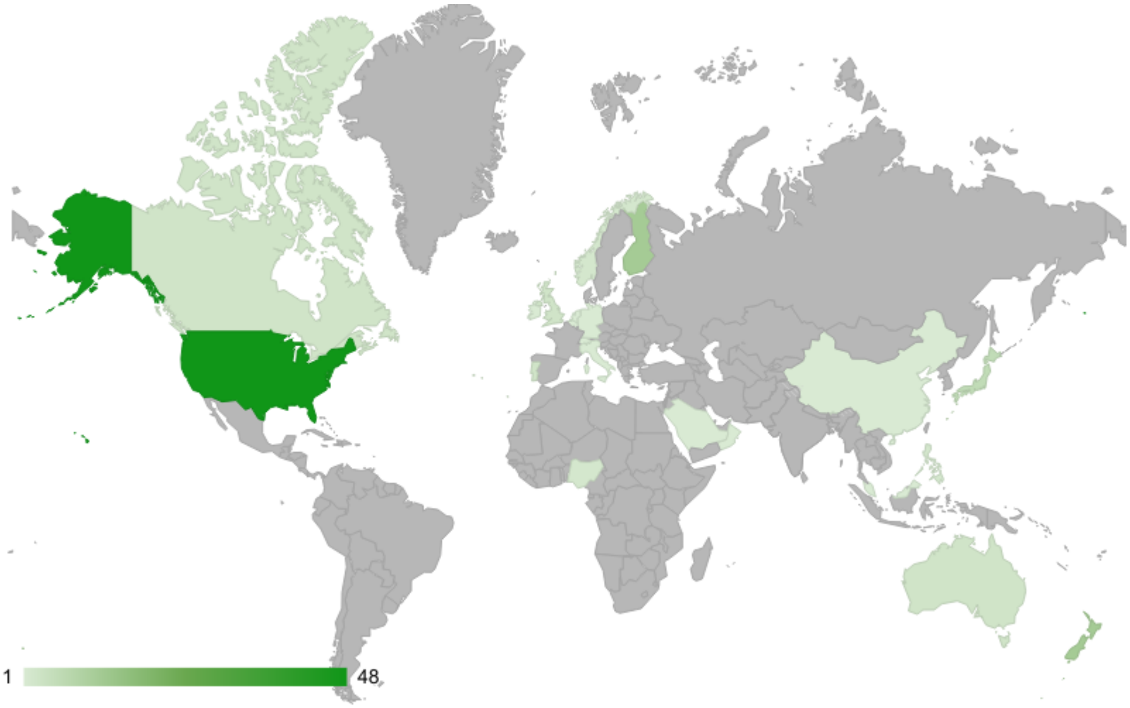


Figure 8: An overview of the countries where studies included in the literature review were conducted. Darker green indicates more published work. The map projection was selected based on the ITiCSE'22 conference location Dublin, Ireland.

tools specifically for delivering novel variations of Parsons problems and as embedded activity modules within online learning platforms.

The original implementation of Parsons problems described by Parsons and Haden in 2006 [177] used an exercise-authoring platform called Hot Potatoes developed at the University of Victoria in Canada. This provided support for the creation of generic drag-and-drop exercises which would present items in a random order to be unscrambled. Of note, when describing future directions for their work, Parsons and Haden described the need for an improved user interface that supports color and animation, as well as a tool that could collect more detailed analytics on student use. The following year, Garner described an evaluation of a tool called CORT, a Visual Basic implementation for supporting 'part-complete solutions' where missing statements (on the left) can be inserted into a part-complete solution (on the right) and then moved around (up or down) within that solution[84]. This 2007 paper does not cite the Parsons and Haden paper but does cite two earlier studies by Garner (including one from 2003) which pre-date the original work on Parsons problems.

Arguably the most influential technology for delivering Parsons problems has been js-parsons (originally stylized as JSParsons), an open-source JavaScript widget developed by Ihantola and Karavirta [109]. The original paper describing the js-parsons tool was also the first to propose two-dimensional Parsons problems, inspired by Python, which required individual lines to be indented correctly. The ease with which problems could be created and embedded using js-parsons led to its wide use in other platforms, including as the core technology in the MobileParsons tool for mobile environments [119]. Other tools reported in the literature for delivering Parsons problems include specialised software such as Kumar's Epplets [128] which provides line-based feedback. Block-based programming environments such as Snap*!* and Scratch [145] have also been extended to support Parsons problems.

Several special-purpose tools have been reported for delivering novel variations of Parsons problems. For example, Weinman, Fox and Hearst describe a Flask app that extends js-parsons for supporting Faded Parsons problems [225], and Pustulka et al. present a custom game-based learning tool called SQL Scrolls, built in Node.js, where the Parsons problems represent SQL queries [189]. The Nester tool, described by Park et al., is novel in our data set in that the lines of code that make up the problems are presented in order, so do not require any vertical movement, but only need to be properly indented [175]. The goal of these problems is to help students develop fluency with hierarchical rules in languages such as HTML, XML, and JSON. In their study, participants completed a reasoning test first (given code, they were asked to identify various elements like siblings and children). Strong negative correlations were found between performance on this reasoning task and time spent solving the Nester task and attempts to solve the Nester task.

Given the increasing popularity of Parsons problems as a pedagogical approach to teaching programming, several Smart Learning Content (SLC) [30] providers introduced ways to present and evaluate Parsons problems within their platforms. Some works took advantage of existing platforms designed for other purposes to deliver Parsons problems. For example, Chirumamilla and Sindre

[35] analysed the potential use of generic Learning Management Systems (LMS) such as Moodle, Canvas, Inspera, and Blackboard to present Parsons problems. Chirumamilla and Sindre show that, while it is possible to use such generic LMS without adding any specific Parsons tooling, creating and using Parsons problems on them is cumbersome and time-consuming.

To ease the workload to introduce Parsons problems, several works created or adapted specialized apparatus. Some SLC eBooks such as Runestone Academy [63], PrairieLearn [226], ICSE Books [41], and Isaac CS [220] integrated Parsons problems along with other programming tasks and content for course delivery (see Table 8), in some cases making use of existing Parsons Technologies such as js-parsons [118] or Epplets [11].

Category	Tool	Papers
SLC	CS Circles	[188]
SLC	Crescendo	[222]
SLC	EvoParsons	[11, 87]
SLC	ICSE Books	[41]
SLC	Isaac CS	[220]
SLC	LMS (Moodle, Canvas, etc.)	[35]
SLC	MasteryGrids	[193]
SLC	PCEX	[105]
SLC	PILeT	[6, 7]
SLC	PrairieLearn	[226]
SLC	REVEL	[41]
SLC	Runestone	[54, 56, 60, 63, 64, 176, 231]
SLC	ViLLE	[118]
PT	BBE (Snap*!*, Scratch)	[33, 145]
PT	Epplets	[127, 128]
PT	Hot Potatotes	[177]
PT	js-parsons	[101, 107, 109, 152, 202, 203]
NV	Faded Parsons	[225]
NV	Nester	[175]
NV	SQL Scrolls	[189]
MD	Mobile Parsons	[107, 119]
MD	MobileEdu	[169, 170]
MoP	MCQ Bubble sheets	[97]
MoP	Paper strips	[162]

Table 8: Tools for Parsons problems. Categories: Smart Learning Content (SLC), Parsons Technology (PT), Novel Variations (NV), Mobile Delivery (MD), and Manual on Paper (MoP).

While the literature is replete with examples of digital tools for delivering Parsons problems, paper-based delivery has continued to be popular for use in exams. In 2008, the first paper that explored the use of Parsons problems on exams was published. The paper, 'Evaluating a New Exam Question: Parsons Problems' by Denny, Luxton-Reilly, and Simon [43], had students solve problems on paper by writing in full each line that they selected from the set of options. In 2021, Stephenson and Mangat describe the use of special-purpose bubble sheets to enable efficient grading at a large scale in paper-based exams [206]. Morin et al. also describe a novel, tangible activity for groups of students where physical strips of paper must be reordered [162].

Tags	Tag Description	Count
LP	Learning Programming	134
RSPF	Research Study Parsons-Focused	64
RSNPF	Research Study but Not Parsons-Focused	56
SP	Student Perception	29
SE	Student Engagement	14
IP	Instructor Perceptions	13
PSSP	Problem Solving Solution Path	12
PSS	Predicting Student Success	11
MD	Mobile Device	11
CL	Cognitive Load	10
UI	User Interface	10
IS	Interventive Scaffolding	8
KT	Knowledge Transfer	8
NNN	Novices vs Near-Novice Learning	6
PPSS	Parsons to Teach Problem Solving Strategies	6
LG	Learning via Gamification	5
EB	Expert Behavior	4
EA	Evolutionary Algorithms	4
GI	Gender Identity	3
GPP	Generating Parsons Problems	3
CSP	Collaboratively Solving Problems	2
LR	Literature Review	2
SAS	Skill Acquisition Sequence	2

Table 9: An overview of the different research question themes ordered by decreasing counts.

4.3 RQ2: What research questions have driven the existing literature on Parsons problems?

Category analysis with category tagging was used to identify and categorize research question themes in the literature. This was done with two researchers who agreed on both the categories as well as each paper's tags. Altogether, 23 research themes were identified with the theme occurring in more than a single article. Each article was tagged with every research theme present in the paper, resulting in each article being tagged with between one and seven tags. These research themes are summarized in Table 9 where they are listed in descending order of occurrence. (Note that the themes identified in the RQ2 research questions are distinct from the research evidence provided, which is discussed in Section 4.4).

For coherence and to facilitate understanding, the discussion on research themes is primarily organized by the similarity of research theme rather than alphabetically or strictly in decreasing order of research theme frequency.

4.3.1 Articles directly related to learning to program. It should not be surprising that the majority of papers studied learning programming (LP) since that was the domain of the inaugural article [177]. Indeed, the vast majority (134 of the 141) of the existing articles on Parsons problems relate to the learning of programming. As discussed earlier, the seminal work by Parsons and Haden is only the third most cited article in the literature. Their key research question is whether they could create an effective tool to allow students to focus on code writing by rearranging lines of code including distractors into a correct code order that solves a given problem. They presented the first documented tool for Parsons problems using exercises designed with the following in mind: maximize the engagement, constrain the logic, permit common errors, model good code, and provide immediate feedback. Their puzzles were drag and drop style and sometimes had an activity chart to lay out the flow in a UML style. They also studied students' perceptions of the activity, finding that the students found the puzzles useful both for learning and in preparing for exams. Because of these research questions, in addition to the LP tag, this article was also tagged as a research study that was Parsons-focused (RSPF) and with student perceptions (SP).

The most cited paper in the Parsons problem literature is by Grover et al. who created and tested an introductory CS course at the middle school level with a focus on computational thinking and algorithmic problem solving [91]. Some of their quiz questions were jumbled Scratch code questions where students must snap the Scratch blocks together in the correct order. Hence Parsons problems were utilized as a part of their research endeavor rather than as a primary focus of their research questions. They studied three issues: 1) the variation across learners in learning of algorithmic flow of control constructs, 2) whether or not students had learning gains in their depth of understanding of algorithmic concepts that went deeper than tool-related syntax details. Here they specifically focused on pedagogical strategies for knowledge transfer from block-based to text-based programming, and 3) whether or not there was any change in students' perceptions of the discipline of Computer Science. In addition to the LP tag, this article was also tagged as a research paper that was not Parsons-focused (RSNPF), knowledge transfer (KT), and students' perceptions (SP). This work is considered notable as the first online introductory middle school curriculum that has been empirically shown to result in learning gains.

The second most heavily cited paper in the Parsons problem literature is by Lopez et al., which can be summarized by the title: "Relationships between reading, tracing and writing skills in introductory programming" [142]. In particular, they utilize a classroom-based study to understand whether the skills of code tracing or code reading (as inferred by "explain in plain English" questions) are associated with program writing ability and whether or not student performance on an exam is consistent with a hierarchy of programming-related skills. The study was not specifically focused on Parsons problems, nor specifically on knowledge transfer, but rather on the correlations between different programming skills. For this reason, this paper was tagged with both LP and RSNPF (research study that is not Parsons-focused).

All eight of the articles tagged with knowledge transfer (KT) studied knowledge transfer during learning to program. Another exemplary article from the KT category is "Precursor skills to writing code" [233]. In this work, the researchers study 1) whether or not students who show proficiency in a given skill also possess proficiency in the skills that precede it in the sequence, and 2) whether or not students who show a deficiency in, or lack, a given skill, also lack the skills that follow it in the sequence. They obtained results suggesting that code comprehension, code manipulation, and code writing are phases that students should sequentially master in the process of learning computer programming. This article was tagged with learning programming (LP) research study that is Parsons-focused (RSPF), knowledge transfer (KT), and skill acquisition sequence (SAS).

Only one other article from the LP literature specifically explored the skill acquisition sequence (SAS), which we mention here in order to distinguish this category from the KT category. The category was intended for those papers that look specifically into the effect of the ordering of skill acquisition. Of course, these might or might not specifically also research the transfer of knowledge from one domain to another. In "Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study" [80], researchers looked into the sequencing of skill acquisition and concluded that optimal order of instruction should be studied directly rather than via any skills hierarchies, so this article was tagged with LP, RSPF, and SAS, but not with KT.

Most of the problems in the LP category either use or study Parsons problems as a part of the content delivery. Some like [177] and [217] use or study Parsons problems as in-class activities. Others leverage Parsons problems via an interactive textbook [60, 101, 107] as a part of the learning design. Still, others (such as [43, 134, 175]) utilize Parsons problems in the assessment of learning. Most of this literature uses traditional Parsons problems. However, more recently, some authors have utilized newer variants such as faded Parsons problems [225].

4.3.2 *Articles not directly related to learning to program.*

It is interesting to see Parsons problems being applied outside of their original domain even within the domain of computing. Some of these articles are focused on a result that is only tangentially related to the learning of programming. For example, some articles focus on developing curricular materials in some unusual way rather than more directly on the learning of programming [85, 87] or on the development of an effective assessment instrument [134]. Other articles focus on identifying patterns in the problem-solving solution path (PSSP) taken as a learner attempts to solve a problem without also specifically researching learning of programming (see for example [148]). Still, others focus on the patterns in the solution path taken by novices vs those taken by experts [109]. Some of these articles do not specifically research the learning of programming but instead study the user interface (UI) of different tools [87]. Of course, manuscripts that are literature reviews generally do not directly research learning programming [48, 195].

4.3.3 *Research Study Parsons-Focused (RSPF).*

We tagged research studies that were specifically focused on Parsons problems as RSPF. These were distinguished by being research studies specifically focused on the utilization and effect of using Parsons problems. Other research studies that utilized Parsons problems, but did not specifically study the effect of the usage then were tagged with RSNPF. Slightly under half of the papers (64 out of 141) included in the literature review were RSPF as opposed to, for example, utilizing Parsons problems in their research while studying something else. An example of a research study that is Parsons-focused is "Mnemonic variable names in Parsons puzzles" by Kumar [130], where the author performed a controlled study comparing student performance with Parsons problems with single-character variable names and Parsons problems with mnemonic variable names. Surprisingly, they found no statistically significant differences in student performance as measured by performance score of the problem, steps or time taken, or time per step.

The three most common themes present in RSPF-papers were learning programming (LP, 60 of 64), student perceptions (SP, 16 of 64), and problem solving solution paths (PSSP, 11 of 64). One example of a RSPF-paper that researched both student perceptions and the problem-solving solution path is "Estimating Learner's Perspective in Programming: Analysis of Operation Time Series in Code Puzzles" by Ito et al. [114] where the authors analyzed time series data from students' solving a Parsons problem in an attempt to identify struggling students. An example of a RSPF-paper that researched student perceptions and also cognitive load (CL) was "Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code" by Haynes and Ericson [98] which found that while most undergraduate students reported that solving adaptive Parsons problem helped them learn, 30% would rather write the equivalent code. These researchers have since revised the Runestone e-book system to give students the choice to solve a Parsons problem or write the equivalent code.

There were a few themes that none of the RSPF-papers included. None of the RSPF-papers studied gender identity (GI) or collaborative problem solving (CSP). In addition, two categories of research questions – "literature review" and "research study but not Parsons-focused" were mutually exclusive with the RSPF tag.

4.3.4 *Research on student or instructor identity, experience, or perceptions.*

A proportion of the Parsons problems literature focuses on cognitive load (CL), student engagement (SE), student perceptions (SP), and/or instructor perceptions (IP) as part of the research questions. Many (29 of 141) of the existing articles study student perceptions (SP), nearly all of which (28 of 29) are also studying LP. For example, some researchers investigated themes such as novices' perceptions of the value of differing instructional formats [94], while others explored themes such as differences in student perceptions of collaborative activities across different student groups or over time [22].

A smaller set of the literature (13 of 141) studied instructor perceptions (IP). For example, Shuhidan et al. investigate instructor perceptions of assessment and their perception of the correlation between assessment and student performance [200]. Four of these 13 articles studied both SP and IP as part of their research focus.

Some of the literature focuses on student engagement (SE), namely (14 of 141) articles, and all of these are also focused on LP. For example, some researchers have explored how to use innovative ideas like the use of hip hop music in teaching coding and how its use affects student engagement [143]. Others proposed new techniques to improve student engagement. Consider Hosseini et al., who focus on how to better support students' acquisition of programming skills through worked examples and the effect of interactivity [104]. Al-Sakkaf et al. have used Parsons problems as part of work to improve student engagement in program visualization using such approaches as the social worked-examples technique [3].

A small proportion of the Parsons problems literature (10 of 141) focuses on researching cognitive load (CL), namely the amount of mental effort necessary to complete a code puzzle, and most of these (7 of 10) are Parsons-focused research papers. For example, Kelleher and Hnin utilize Parsons Problems heavily in the creation of a model for predicting cognitive load [123].

Two different tags relate to identities of expertise. Six articles compare novices to near-novices (NNN), all of which are in the domain of learning programming (LP). One example in this category is "Representing and Evaluating Strategies for Solving Parsons Puzzles" which develops a proof of concept representation for solution strategies of students in data collected by a Parsons problem [131]. Four of the articles report on expert behaviors (EB), all of these using a control group for comparison. Some of these compare behaviors of novice programmers versus more advanced programmers. Of these, only "Investigating strategies used by novice and expert users to solve Parsons problems in a mobile python tutor" utilizes a research study that is Parsons-focused (RSPF), confirming that experts used superior problem-solving strategies [67]. Another explored how teachers and students engage with a course content in an e-book differently [176].

A surprisingly small number of articles (3 of 141) explicitly explore gender identity issues as part of the research questions. For example, "Speed and Studying: Gendered Pathways to Success" considered whether female students are more conscientious in starting assignments earlier, completing their problem sets, or spending more time studying [230]. Lai et al. research the intersection of programming experience with gender as part of an assessment instrument on computational thinking performance [134]. Becker et al. study whether or not extra credit serving as a performance differentiator applies to both women and men similarly [17].

4.3.5 *Articles related to pedagogy.*

Several articles in the Parsons problems body of literature have research questions focused on various aspects of pedagogy. Six articles explicitly study using Parsons problems to teach problem-solving strategies (PPSS). This seemed such an important use of Parsons problems that it earned its own own tag. An exemplar in this category is an article that evaluates the effectiveness of Parsons problems for block-based programming, finding that Parsons problems save students nearly half of total problem solving time [234]. Another exemplar in this category is an article that looks at Parsons problems as a design-based intervention as well as at how self-regulated learning strategies are applied by students when interacting with Parsons problems [82]. Two articles have research questions related to collaboratively solving problems (CSP), including team-based learning [22] and using teams during distance learning [146].

4.3.6 *Articles using Parsons problems as scaffolding interventions.*

Eight of the papers in our literature review include research questions that involve an intervention during the problem solving process. The interventions include giving feedback, limiting feedback, visualizing the solution path, and encouraging students to make selections. This work was tagged with interventive scaffolding (IS). One paper [119] studies how to provide meaningful feedback on Parsons problems in a mobile application, with feedback disabled for short periods of time when the frequency of feedback requests gets high. In another study [82], the Parsons problems are not code but rather made of goals and tasks. Students use the feedback from these Parsons problems in the design process of problem solving. One study [101] records a detailed trace of all the interaction in solving a Parsons problem that provides insights into students' problem solving process. The same authors build on the interaction traces in a follow-up paper [100], where they compare two types of feedback from two groups of students, execution-based and line-based. With the execution-based, feedback was requested less frequently. A tool paper [105] focused on worked-out program examples and engages students with challenge activities the students must initiate. The challenge activities are similar to the worked-out examples and could be Parsons problems.

A paper with block-based programming [234] provides a check button with their Parsons problems only after student code has been run a few times. If code is not correct, then it will highlight misplaced code. One paper [202] extended previous feedback with visualizations of program execution and observed more than half of the students viewed the visualizations if they were available. In another paper [72] Parsons problems were one of many activities for students to choose from in a study using a tool that focused on several coding skills. Their study showed their tool was effective for learning such skills.

4.3.7 *Research related to the problem solving solution path (PSSP).*

There were twelve papers that were identified by the research theme "problem solving solution path" (PSSP) that focuses on the steps learners take in getting to a solution for a Parsons problem. One paper looks at the common patterns students use when solving a Parsons problem, such as a linear approach (moving the blocks in the solution order), and identified difficulties students have such as looping (returning to a previous state) [101]. Another paper took all the student paths for solving a specific Parsons problem and modelled them in state transition diagrams for use by instructors and teaching assistants to see trends to help them with instruction [217]. The state transition diagram for correct answers was reasonable, but the state transition diagram for wrong answers was somewhat unwieldy.

One article focused on two approaches for helping students solve Parsons problems [129], firstly, researching pairing distractors with correct line of code instead of randomly showing them, and secondly, researching the timing of penalizing the student, namely while solving a Parsons problem versus penalizing them after submitting a complete solution. Another paper has a component with a process called Use-Modify-Create for scaffolding the learning process [143]. Parsons problems are one part of that process. Ericson et al. explored adaptive Parsons problems, where the user can ask for help while solving the problem, comparing the number of extra steps taken to get to a Parsons problem solution, with the time to solve the problem [61]. Other researchers analyze Parsons problem solutions using a Markov Transition Matrix [133]. These researchers only analyzed correct and complete solutions, and they found several patterns such as 1) most students build the solution in order, 2) most students discard distractors early and 3) most students start with the variable declaration.

Another paper estimates the learner's solution path by using logs from users solving Parsons problems, estimating the learner's perspective with a Markov model [114]. One paper investigates three strategies for solving a Parson problem including 1) an approximate linear representation of their linear behaviour 2) A BNF grammar to represent their strategies and 3) a best-match parser [131]. Their approach can be used for any programming language. Another paper describes a tool called Epplets that gives feedback to students as they solve Parsons problems [128]. Practicing with the tool got students to solve Parsons problems faster. One paper uses edit distances to analyze the solution paths of users solving Parson problems to identify patterns [149]. Another paper analyzes the edit distance between the student solution for a Parsons problem and the correct solution, looking at each step the student takes [148]. Ihantola et al. observe experts in solving Parsons problems as they use the jsParsons tool, finding that experts do not ask for much feedback from the tool and do not solve the problems linearly [109].

4.3.8 *Research on Parsons problems delivery.*

A subset of the Parsons problems research focuses on some specific aspect related to the delivery of Parsons problems. Eleven focus on issues resulting from delivery on a mobile device (MD) and are tagged with MD. Ten articles look at the user interface (UI) specifically, and are tagged with UI. Four articles are tagged with both MD and UI because they are focusing on UI on a mobile device. The article [225] is an example of an article that focuses specifically on the effect of the user interface on knowledge acquisition, but is not utilizing a mobile technology. The work in [169] does not look at the user interface, but expands a mobile learning system to include a traditional African strategy board game with Parsons problems in order to study student interaction, motivation, and engagement during the process of learning to program. The work in [119] looks specifically at how to adapt Parsons problems to mobile devices, so has both the MD and UI research themes.

4.3.9 *Research related to learning via gamification.*

There are five papers that focus on research on learning via gamification. Four of them built a computer game in which Parsons problems were a game activity, and the last one creates a gamelike atmosphere in class. The first paper [189] built an online game for students to learn about databases and SQL that includes solving SQL Parsons problems. The second paper [2] developed a gamified recommender system to provide recommendations to motivate students as they learn programming, with Parsons problems as puzzles in the system. The third paper [47] created a three-part computer game for learning about stacks for a data structures course, with Parsons problems to solve in the third part of the game. The fourth paper [169] has a mobile app for a board game and another mobile app for Parsons problems, and they are laying the framework to combine the two, where a player would need to solve a Parsons problem on coding when it is their turn to play. The fifth paper [136] is not a computer game, but uses game tactics in a flipped CS1 course. Students in the course work in teams and solve tasks with game-like elements such as stamps, a leaderboard and question tokens. Parsons problems are done as some of the activities.

4.3.10 *Categories with very few papers.*

There were several categories that had only a few papers. We describe those categories and corresponding papers here.

Two papers were categorized as a literature review (LR). One of them was a literature review of Parsons problems focused on three research questions. We describe that paper in more detail in Section 2.4 [48]. The other was a literature review of insights into learning issues such as effective learning tasks and teaching methods [195].

Four papers were categorized as evolutionary algorithms (EA) and all of them looked at automatically generating Parsons problems (GPP) [11, 12, 85, 86].

There were five categories that had only one paper. 1) Only one paper was categorized as at-risk students, and focused on identifying potentially at-risk introductory programming students [17]. Their study used Parsons problems as one of many extra credit opportunities. 2) One paper was categorized as meaningful variable names, and focused on whether or not mneomonic variable names made it easier to solve Parsons problems than single letter variable names. They looked at time to solve and number of line movements, finding no significant difference between using mnemonic variable names versus single letter variable names [130]. 3) One paper was categorized as novices creating curricular material, and was focused on novices trained to create curriculum material for K-12 students [33]. In particular, high school students in a summer internship program were able to create quality programming lessons for non-computing courses. One result was that Parsons problems were one of the easier types of lessons to create. 4) One paper was categorized as using the process of notional machines. The paper compared sketching out code tracing to no sketching when solving different types of problems [38]. One of their findings was that very few students (3%) sketched out code ordering problems (these are Parsons problems), yet both those who sketched and those who did not sketch all scored high in these types of problems (over 96%). The researchers state that the small amount of sketching could indicate that Parsons problems have low cognitive load. 5) One paper was categorized as validation of an assessment instrument, and presented a validated instrument to measure computational thinking competency [134], of which Parsons problems are one of the types of problems on the assessment.

4.3.11 Most common jointly reoccurring research themes. We studied which research themes tended to occur frequently together by analyzing research themes that were jointly present. The most common research themes were present jointly with Learning Programming, which was also the most common theme as discussed above. Out of the 134 articles that looked into Learning Programming, 60 were Parsons-focused research studies (RSPF), 54 reported on a research study that was not Parsons-focused (RSNPF), and 28 articles looked into Student Perceptions (SP). Out of the total of 29 articles that looked into Student Perceptions, almost all (28/29) were also related to learning programming (LP), 16 presented a research study that was Parsons-focused (RSPF), and 11 were related to not Parsons-focused studies (RSNPF). The jointly occurring research themes are summarized in Figure 9.

4.4 RQ3: What evidence exists for the claimed benefits that Parsons problems offer, and for their limitations, and what is the quality of that evidence?

We examine the quantitative measurements presented in each paper, to provide insight not only into what types of evidence exist but also how strong that evidence is. This will provide researchers with guidance as to areas where future work is needed. Table 10 shows the types of evidence being presented in the papers in our data set.

Figure 10 illustrates the range of study sizes, by number of participants, for each of the evidence types we have identified. Most types of evidence presented still require large-scale replications to verify their findings. Some types of evidence, such as using Parsons to motivate student learning or using Parsons to understand errors students make, have only been studied in relatively small populations. Curiously, most studies that use data on students solving Parsons problems to feed into a machine learning algorithm have relatively low sample sizes. These represent clear avenues of future work.

Evidence type	Frequency
learning gains	22
engagement	17
student analysis	15
code writing	11
speed	11
cognitive load	9
code tracing	7
predictive	7
student perceptions	7
motivation	6
none	6
parsons analysis	6
subgoals	4
errors	4
ease of grading	3
struggle parsons	3
feedback	2
help-seeking	2
hierarchies	2
misconceptions	2
patterns	2
benefit non-majors	1
computational thinking	1

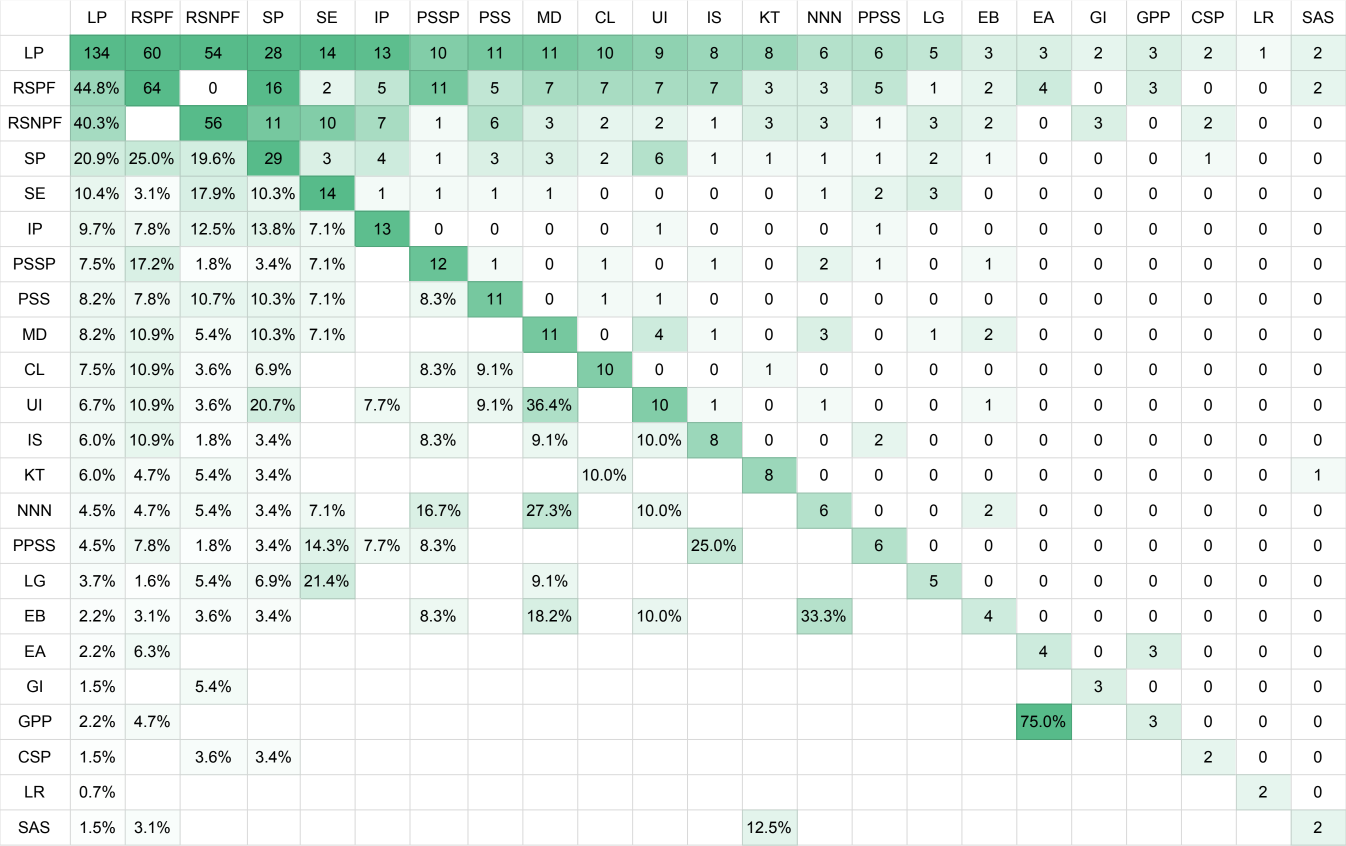
Table 10: Type of evidence being presented in papers

Figure 11 visualizes the types of research evidence that typically appear together in Parsons problems related papers. From the figure, one can see that most commonly, only a single type of evidence is presented.

Looking at co-occurring evidence, one interesting finding is that evidence related to cognitive load is almost always accompanied by some other type of evidence, although the type of that other evidence in these papers is very varied – in fact, every other type of evidence except for subgoals appeared at least once together with cognitive load.

Looking at the percentages in the lower diagonal of Figure 11, one can see that percentage-wise, the two most commonly jointly appearing evidence types are code writing and code tracing: 36.4% of papers that had evidence related to code writing also had evidence related to code tracing.

4.4.1 Learning Gains. We tagged 22 papers that presented evidence on learning gains with Parsons problems [6, 20, 31, 40, 59, 61, 62, 68,



	LP	RSPF	RSNPF	SP	SE	IP	PSSP	PSS	MD	CL	UI	IS	KT	NNN	PPSS	LG	EB	EA	GI	GPP	CSP	LR	SAS
LP	134	60	54	28	14	13	10	11	11	10	9	8	8	6	6	5	3	3	2	3	2	1	2
RSPF	44.8%	64	0	16	2	5	11	5	7	7	7	7	3	3	5	1	2	4	0	3	0	0	2
RSNPF	40.3%		56	11	10	7	1	6	3	2	2	1	3	3	1	3	2	0	3	0	2	0	0
SP	20.9%	25.0%	19.6%	29	3	4	1	3	3	2	6	1	1	1	1	2	1	0	0	0	1	0	0
SE	10.4%	3.1%	17.9%	10.3%	14	1	1	1	1	0	0	0	0	1	2	3	0	0	0	0	0	0	0
IP	9.7%	7.8%	12.5%	13.8%	7.1%	13	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
PSSP	7.5%	17.2%	1.8%	3.4%	7.1%		12	1	0	1	0	1	0	2	1	0	1	0	0	0	0	0	0
PSS	8.2%	7.8%	10.7%	10.3%	7.1%		8.3%	11	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
MD	8.2%	10.9%	5.4%	10.3%	7.1%				11	0	4	1	0	3	0	1	2	0	0	0	0	0	0
CL	7.5%	10.9%	3.6%	6.9%			8.3%	9.1%		10	0	0	1	0	0	0	0	0	0	0	0	0	0
UI	6.7%	10.9%	3.6%	20.7%		7.7%		9.1%	36.4%		10	1	0	1	0	0	1	0	0	0	0	0	0
IS	6.0%	10.9%	1.8%	3.4%			8.3%		9.1%		10.0%	8	0	0	2	0	0	0	0	0	0	0	0
KT	6.0%	4.7%	5.4%	3.4%						10.0%			8	0	0	0	0	0	0	0	0	0	1
NNN	4.5%	4.7%	5.4%	3.4%	7.1%		16.7%		27.3%		10.0%			6	0	0	2	0	0	0	0	0	0
PPSS	4.5%	7.8%	1.8%	3.4%	14.3%	7.7%	8.3%					25.0%			6	0	0	0	0	0	0	0	0
LG	3.7%	1.6%	5.4%	6.9%	21.4%				9.1%							5	0	0	0	0	0	0	0
EB	2.2%	3.1%	3.6%	3.4%			8.3%		18.2%		10.0%			33.3%			4	0	0	0	0	0	0
EA	2.2%	6.3%																4	0	3	0	0	0
GI	1.5%		5.4%																3	0	0	0	0
GPP	2.2%	4.7%																75.0%		3	0	0	0
CSP	1.5%		3.6%	3.4%																	2	0	0
LR	0.7%																					2	0
SAS	1.5%	3.1%											12.5%										2

Figure 9: Jointly occurring research themes in Parsons problems related papers. The upper diagonal shows the number of papers that had a specific research theme pair co-occur. The lower diagonal shows how often the research theme on the left co-occurred with the research theme at the top as a percentage. For example, considering LP and RSPF (the two most common themes), 60 papers tagged with LP were also tagged with RSPF, leading to 60/134 = 44.8%. The empty cells in the lower diagonal represent 0%. Please note that a paper could be tagged with multiple different research question themes.

70, 71, 81, 95, 96, 104, 105, 116, 117, 127, 147, 178, 231, 234]. Some papers compared learning gains from solving different types of Parsons problems such as Parsons problems with self-explanation versus not [70, 71], with distractors versus without [95], and with motivational support and without [127]. Other papers compared solving Parsons problems to other types of practice including tutorials [96], fixing textual code problems [62], writing textual code [59, 62], or assembling blocks-based solutions [20, 234]. Many of the papers measured learning gains from systems that used Parsons problems along with other types of practice and did not isolate the contribution from solving Parsons problems [6, 40, 68, 81, 104, 105, 116, 117, 231]. Two papers only included Parsons problems in the assessment [81, 178].

Research methods in this category were quantitative (20), qualitative (6), and both (5), with the vast majority presenting clear research questions and a reproducible methodology. Most of the studies (16) had two or more groups in their studies. However, only half of the studies had clearly detailed threats to validity listed. The studies range from 27 to 736 participants and mostly focused on Python (9) and Java (8) with one or two studies in C++, Scratch, VPL, and Looking Glass. A wide variety of types of Parsons problems are known to provide learning gains to students, including basic, faded, adaptive, and with distractors. However, no studies presented evidence of learning gains with mobile Parsons. Most of the studies were on-campus (9), one was online, and the rest did not mention delivery type. Overall, evidence in this category is high quality.

4.4.2 Engagement. In total, 17 papers had evidence of student engagement [11, 31, 60, 86, 88, 90, 94, 104, 111, 127, 129, 161, 176, 177, 222, 229, 231]. Parsons and Haden reported that 82% of their undergraduate students found Parsons problems useful or very useful for learning [177]. Morin and Kecskemety reported that 60% of their undergraduate students were positive about Parsons problems for C++ and Matlab. Ericson, Guzdial, and Morrison found that more students tried to solve Parsons problems in Python than nearby multiple-choice questions in a free and interactive ebook [60]. However, some learners have a strong negative reaction to solving Parsons problems [98].

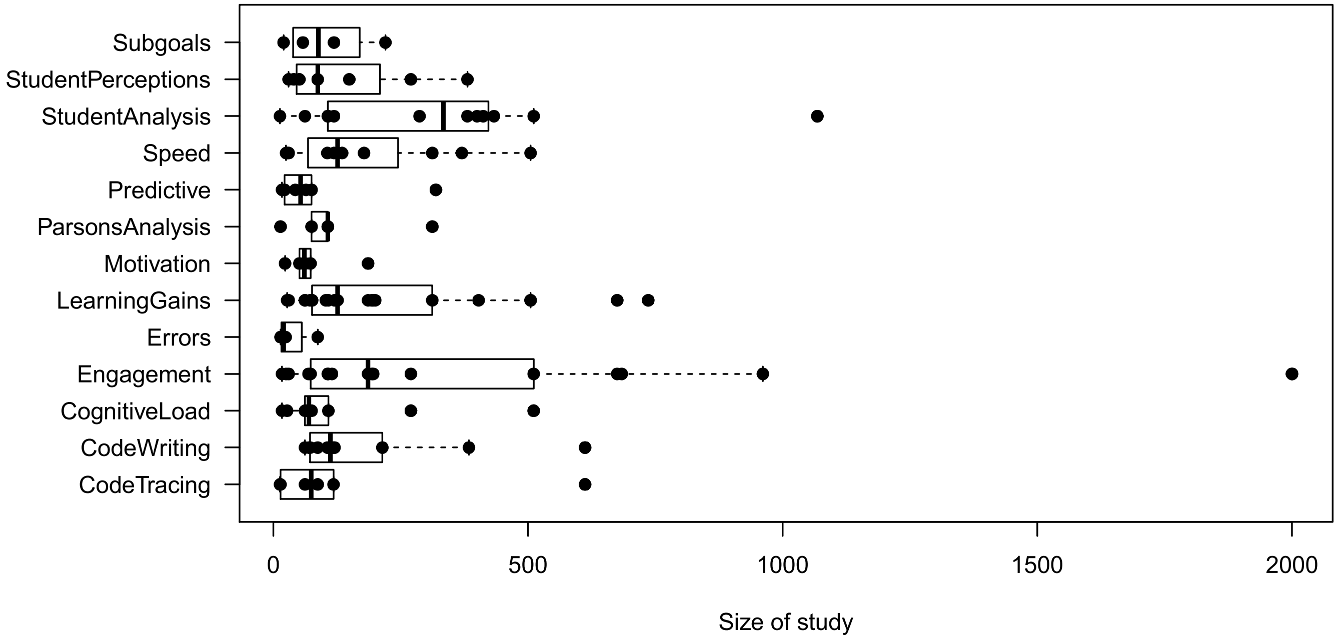
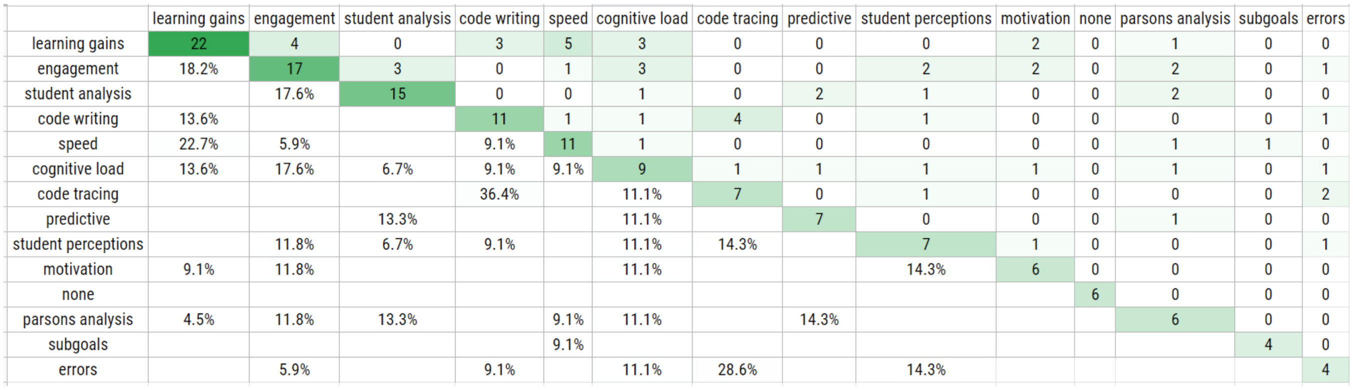


Figure 10: Number of participants in studies by evidence type



	learning gains	engagement	student analysis	code writing	speed	cognitive load	code tracing	predictive	student perceptions	motivation	none	parsons analysis	subgoals	errors
learning gains	22	4	0	3	5	3	0	0	0	2	0	1	0	0
engagement	18.2%	17	3	0	1	3	0	0	2	2	0	2	0	1
student analysis		17.6%	15	0	0	1	0	2	1	0	0	2	0	0
code writing	13.6%			11	1	1	4	0	1	0	0	0	0	1
speed	22.7%	5.9%		9.1%	11	1	0	0	0	0	0	1	1	0
cognitive load	13.6%	17.6%	6.7%	9.1%	9.1%	9	1	1	1	1	0	1	0	1
code tracing				36.4%		11.1%	7	0	1	0	0	0	0	2
predictive			13.3%			11.1%		7	0	0	0	1	0	0
student perceptions		11.8%	6.7%	9.1%		11.1%	14.3%		7	1	0	0	0	1
motivation	9.1%	11.8%				11.1%			14.3%	6	0	0	0	0
none											6	0	0	0
parsons analysis	4.5%	11.8%	13.3%		9.1%	11.1%		14.3%				6	0	0
subgoals					9.1%								4	0
errors		5.9%		9.1%		11.1%	28.6%		14.3%					4

Figure 11: Jointly occurring research evidence in Parsons problems related papers. The upper diagonal shows the number of papers that had a specific research evidence pair co-occur. The lower diagonal shows how often the research evidence on the left co-occurred with the research evidence at the top as a percentage. For example, considering learning gains and engagement (the two most common types of evidence), 4 papers tagged with "learning gains" were also tagged with "engagement", leading to 4/21 = 19.0%. The empty cells in the lower diagonal represent 0%. Please note that a paper could be tagged with multiple different types of research evidence, and that only papers tagged with IC1 (empirical papers) are included.

The most commonly used programming language was Python, present in 7 papers, while the second most commonly used programming language was Java, which was present in 4 papers. In practice, when considering these programming languages, one could consider that the languages themselves might influence engagement, and thus comparisons between the languages would be beneficial. In the majority of cases, engagement was studied quantitatively (14 papers), while 3 of the papers used mixed methods applying both quantitative and qualitative analysis. In general, the quality of the work was reporting-wise good, where the vast majority of the papers clearly reported the research process (14 papers) and reported the results in sufficient detail (15 papers). At the same time, only 6 articles explicitly discussed the limitations of the work.

When considering the study designs, almost half of the studies were conducted in person (8 papers), while 5 of the studies were conducted in an online environment. Of the 17 papers, only 6 papers had two or more groups, allowing for a comparison that would account for at least some of the possible confounding variables. The most commonly studied Parsons problems type was basic Parsons problems (10 papers), while 6 papers also included Parsons problems with distractors. Little work on adaptive Parsons problems or Faded Parsons problems exists.

4.4.3 Student Analysis. A total of 15 papers looked into how students used Parsons problems and into Parsons problems that analyzed students learning and adapted into it [11, 17, 67, 77, 86, 100, 129, 131–134, 148, 149, 193, 217].The most commonly used programming language included variants from the C/C++/C# -family (9 papers) and Java (7 papers). Here, the majority of the analyses are quantitative, looking into e.g. numbers of steps that students take or issues that students face. Similarly to studies on student engagement, the reporting was mostly good: most papers appropriately outlined the research process (10 papers) and sufficiently outlined the research results (10 papers). Threats to validity were explicitly reported rather scarcely (2 papers), although a few papers briefly discussed some issues in the presented studies. A handful of papers presented studies with more than a single population/group (5 papers), which could reflect that a deeper analysis of behavior is often conducted in a single population in a single condition (e.g., a single type of Parsons problem). Similar to the results on engagement, the most commonly used Parsons problems were the basic Parsons problems (in 9 papers) and Parsons problems with distractors (in 7 papers). Likewise, the majority of the studies were conducted in person (7 studies), while for many of the studies the location for data collection was not available or was unclear (5 papers).

4.4.4 Code Writing. There were 11 papers that presented evidence on how Parsons problems impacted programming students' code writing abilities [43, 69, 73, 80, 81, 97, 98, 105, 138, 178, 201]. Most presented clear research questions (7) and presented their research in a clear (9) and reproducible (9) manner. However, only one study explicitly listed clear threats to validity and only three used more than one group. All studies in this group had a quantitative element and three also contained qualitative methods. Python (6) and Java (4) were the most popular languages in this category, with one study each on C, C#, Perl, Pascal, and VPL. The number of participants ranged from 62 to 612, with a majority of the studies with more than 100 participants. Types of Parsons studied in this category are basic (7), faded (2), adaptive (1), and with distractors (4), with no studies on indentation or on mobile contexts. The delivery method was mostly in-person (6) with one online and one hybrid method, and the rest unclear.

4.4.5 Speed. A total of 11 papers provided evidence on problem-solving efficacy in terms of speed and studied changes in it as a consequence of using Parsons problems [59, 61, 62, 72, 90, 96, 98, 128, 165, 167, 234]. The papers provided evidence that solving Parsons problems was significantly faster than completing a tutorial [96], fixing code with errors [62], writing textual code [59, 62], or assembling a blocks-based solution [234]. However, Haynes and Ericson found that a Parsons problem with a solution that did not match the common student solution was not significantly faster to solve than writing the equivalent code [98].

The most commonly used programming languages were Java (2 papers), Python (5 papers), and variants from the C/C++/C#-family (4 papers). In these papers, the research questions were almost always explicitly defined (10 papers), the research process was clear (11 papers), and the results were sufficiently outlined (10 papers). Similar to the earlier categories, limitations of the study was more rarely reported, where only 2 papers explicitly outlined threats to validity or limitations of the study in a separate section of the respective paper. Methodology-wise, the majority of the studies had a quantitative aspect (9 papers), which is to be expected in studies that seek to quantify changes in a metric such as problem-solving speed. Out of the papers in this category, 7 had two or more participant groups, and 6 compared two or more Parsons problem types. Here, 4 papers included distractors and 4 included adaptive parsons problems, while 6 articles featured basic parsons problems – some of the Parsons problem types occurred jointly in the same articles.

4.4.6 Cognitive Load. We categorized nine papers that presented evidence that Parsons problems impacts programmers' cognitive load or that working on Parsons problems has a lower cognitive load than writing code problems of similar difficulty [20, 38, 73, 95, 96, 123, 129, 161, 177]. Papers presenting this type of evidence have clear research questions and methodologies and provide sufficient details to reproduce their experiments. However, only two papers in this category had explicit limitations and only four had more than one group in their study. The number of participants ranges from 27 to 511. Studies were quantitative (7), qualitative (2), or mixed (2). Languages used in these studies were Looking Glass (3), Python (2), C++ (2), and one in each of MATLAB, Scratch, and Pascal. The types of Parsons problems studied were basic (8), with distractors, (6), faded (1), and adaptive (1). These studies were delivered in-person (3) or online (2), with the rest unclear. The evidence that Parsons problems can impact student cognitive load is extremely high quality.

4.4.7 Code Tracing. A total of 7 papers investigated the effect of Parsons problems on students' code tracing ability [5, 38, 43, 57, 69, 80, 138], i.e., the ability to read code and mentally interpret and simulate how the program would work. Similarly to the earlier studies, the most common programming languages were Python (4 papers) and Java (2 papers), with a few studies using the C/C++/C#-family. The studies were quantitative (7 papers), although 2 papers also featured qualitative analyses. Of the 7 papers, 5 of them contrasted two or more Parsons problem types with each other, although having multiple groups of students (e.g., randomized controlled trial) was rare (2 papers). When considering the reporting, 5 out of the 7 papers had clear research questions, provided a clear research process, and outlined the results in sufficient detail. Similar to the previous studies, it was relatively rare to explicitly outline the limitations of the study or threats to validity, which was explicitly discussed in its own part in 2 papers.

4.4.8 Predictive. We identified 7 papers with evidence of predicting students' performance using data from Parsons problems [76, 79, 112, 113, 123, 149, 193]. Interestingly, the majority of the papers in this category were from the C/C++/C#-family (6 papers), while more featured languages such as Python and Java were present in only one of the papers. All 7 papers followed a quantitative analysis, although one of them also had a qualitative component. None of the papers had a research design with two or more groups of students, and the papers also did not contrast between types of Parsons problems. Out of the 7 papers, 6 featured basic Parsons problems, while faded, adaptive, and problems with distractors were present in one paper.

In these studies, on average, the formulation of the study objectives could be improved, as only 3 of the 7 papers explicitly outlined research questions. At the same time, 5 papers did clearly outline the research process, and 6 papers reported the results with sufficient detail. Limitations or threats to validity were explicitly discussed in a separate section in 2 of the 7 papers, continuing the trend observed in other evidence categories.

4.4.9 *Students' Perceptions.*

A total of 7 papers provided evidence of students' perceptions [43, 94, 100, 102, 118, 161, 168], i.e., considering how students felt about using Parsons problems. In contrast to some of the previous categories where the analyses were primarily quantitative, a large proportion of the students' perceptions studies included had a qualitative analysis (5 papers), often accompanied by quantitative analysis (6 papers). Here, 5 of the papers reported results from an in-person study, 1 from an online study, and 1 from a hybrid study. The types of Parsons problems used were unclear in 3 of the studies, while 4 of the studies used basic Parsons problems and Parsons problems with distractors. When considering the number of Parsons problems, in 3 papers the number of Parsons problems used was unclear, while 4 of the papers used five or more Parsons problems.

The research process was sufficiently outlined in 5 of the papers, while 4 papers reported the results in sufficient detail. Only 3 papers explicitly outlined research questions, effectively highlighting the need to make the research objectives more explicit for readers. Similar to the previous categories, explicitly discussing threats to validity or limitations of the study was relatively rare (2 papers).

4.4.10 *Motivation.*

We categorized six papers that presented evidence on how Parsons problems can be used to motivate students to learn programming [2, 20, 127, 168, 189, 222]. Only three of the papers in this category had clear research questions and clearly described the research process. None had clearly defined limitations or threats to validity. Only three of the papers had two or more groups in their study and the only types of Parsons problems studied were basic and basic with distractors. Each paper included in this category uses a different programming language: Java, C++, Visual Basic, SQL, Snap*!*, and Scratch. All six studies contained quantitative measures and three of the studies also included qualitative measures. The number of participants ranged from 23 to 186, with five of the six studies at or less than 73 participants. Studies presenting evidence on motivating students were delivered in-person (2), online (1), hybrid (1), and the rest were unclear. While motivating students to learn programming is a very important outcome to study, stronger evidence is needed.

4.4.11 *Parsons Analysis.*

We categorized as Parsons analysis articles where the authors sought to create new types of Parsons problems and study them. A total of 6 papers fell into this category [11, 12, 86, 123, 157, 234], including e.g., the use of ML techniques to automatically evolve Parsons problems [11, 12] and automatically creating Parsons problems from code [86]. Methodologically, all 6 papers focused on quantitative analyses, while only 3 of the papers compared and contrasted two or more types of Parsons problems. Out of the papers, 5 outlined a clear research process, while 4 explicitly provided the research questions and outlined the research results in sufficient detail.

4.4.12 *Subgoals.*

We categorized four papers that presented evidence on Parsons problems and subgoals [39, 115, 164, 165]. Two papers had clear research questions, and three had a clearly detailed process and reproducible methodology. However, only two had more than one group in the study, and only one of the papers had clearly defined threats to validity. Three of the four studied basic Parsons problems, and one studied basic Parsons problems with distractors. Only Python and C have been used for this type of evidence and the number of participants ranges from 20 to 220. Two of the studies were delivered in-person, one was online, and one was unclear. The quality of evidence in this category is relatively strong despite only having four exemplars. However, much future work remains on the impact of using subgoals in Parsons problems.

4.4.13 *Errors.*

Four papers were identified as studying errors that students make in Parsons problems [43, 57, 175, 177]. The analyses were both quantitative and qualitative in 2 papers, while one paper featured only a quantitative analysis and one paper featured only a qualitative analysis. The research process was clear and the reporting of the results was sufficient in all of the papers, although only 2 out of 4 papers had explicit research questions; similarly, 2 of the 4 papers also had explicitly dedicated parts for threats to validity or limitations of study. Number of participants was relatively small: 14, 17, 24, 87. Languages used include C, HTML, JavaScript, Pascal, and Python. Types of Parsons problems used were basic, adaptive, and distractors. Although the evidence in this category is good, there is still very little work on the matter.

4.4.14 *Other evidence.*

Nine types of evidence had three or fewer papers with that tag. We therefore present them here in a group.

Three papers presented evidence on how Parsons problems can ease the grading load on instructors [34, 43, 201]. While the number of participants in these studies is relatively high and was tested with more than one group, they only tested basic Parsons problems and did so in-person. These studies were conducted in a broad set of international contexts and only in tertiary education.

Three papers presented evidence that students can struggle with Parsons problems if not properly equipped to tackle them before attempting them [60, 67, 114]. The largest study in this group utilized over 2,000 participants and presents high quality evidence on basic Parsons problems in both secondary and tertiary contexts in the USA. The other two studies, while conducted in international contexts of Japan and New Zealand, have a very low number of participants and provide vague details about research methods. More work in this area is needed to confirm the existing evidence in the literature.

Two papers discussed how Parsons problems are useful for providing timely feedback to students about their understanding of the problem [100, 177]. Large and small studies representing quantitative and qualitative data are represented in New Zealand and Finland. However, these studies were only in tertiary contexts on basic Parsons problems. Future work is needed in other countries, in other educational contexts, and with newer variations of Parsons problems.

Two papers looked at students' help-seeking behaviors when working Parsons problems [61, 147]. Both studies provide data from large online quantitative studies in both tertiary and secondary educational contexts in the USA on both basic and adaptive Parsons problems. While these studies present high quality evidence, future work is still needed to confirm and extend their findings.

Two studies used Parsons problems to examine student understandings of hierarchies and relationships in code [142, 175]. Both studies present quantitative evidence with a small (<100) number of participants, studying basic Parsons problems, without utilizing a control and experimental group, in the USA and New Zealand. The quality of evidence on these claims is relatively low and future work is needed to better determine how Parsons problems impact student understandings of hierarchies.

Two papers utilized Parsons problems to get insight into student misconceptions [12, 54]. Both studies present quantitative evidence on the use of Parsons with distractors in a USA context. However, participant counts were relatively moderate (<108) or not reported and research methodology was unclear. These works have exposed a viable avenue of future work on how Parsons problems can be used to understand student misconceptions while learning to code.

Two papers examined how students solve Parsons problems and found common patterns [101, 217]. Both papers presented quantitative evidence on the use of Parsons, one with distractors and one without, in international contexts of Japan and Finland. However, both studies lacked clear research questions and did not contain enough details in their research methods to reliably reproduce their results. Therefore, more work is required to validate evidence on the patterns students use to solve Parsons problems.

One paper presented evidence that Parsons can be a useful teaching tool for non-majors [39], one paper examined how Parsons impacts students' computational thinking [134], and one paper compared execution-based feedback to line-based feedback and found benefits and drawbacks to both [100]. More work is needed in these categories to validate their claims.

Figure 12 shows a cross tabulation between the research themes based on RQ2 and the types of research evidence based on RQ3. From the figure, it is evident that there are some themes that have a substantial amount of evidence of various types while other themes are quite lacking in evidence, thus providing ample opportunities for future research.

For example, learning programming (LP), student perceptions (SP), and research studies focusing on Parsons problems (RSPF) had at least one paper presenting each of the different types of evidence we analyzed. On the other hand, two themes – literature review (LR) and collaborative problem solving (CSP) did not have any papers with evidence. This is not very surprising, however, since there were very few papers falling under those two themes (2 for LR, 2 for CSP), while the themes with a lot of varied evidence were also the most common themes (LP with 134, SP with 79, and RSPF with 64 papers).

5 GAP ANALYSIS AND RESEARCH DIRECTIONS

From the results of our literature review, we conduct a gap analysis related to the three main research questions to identify directions for future research. These are discussed next.

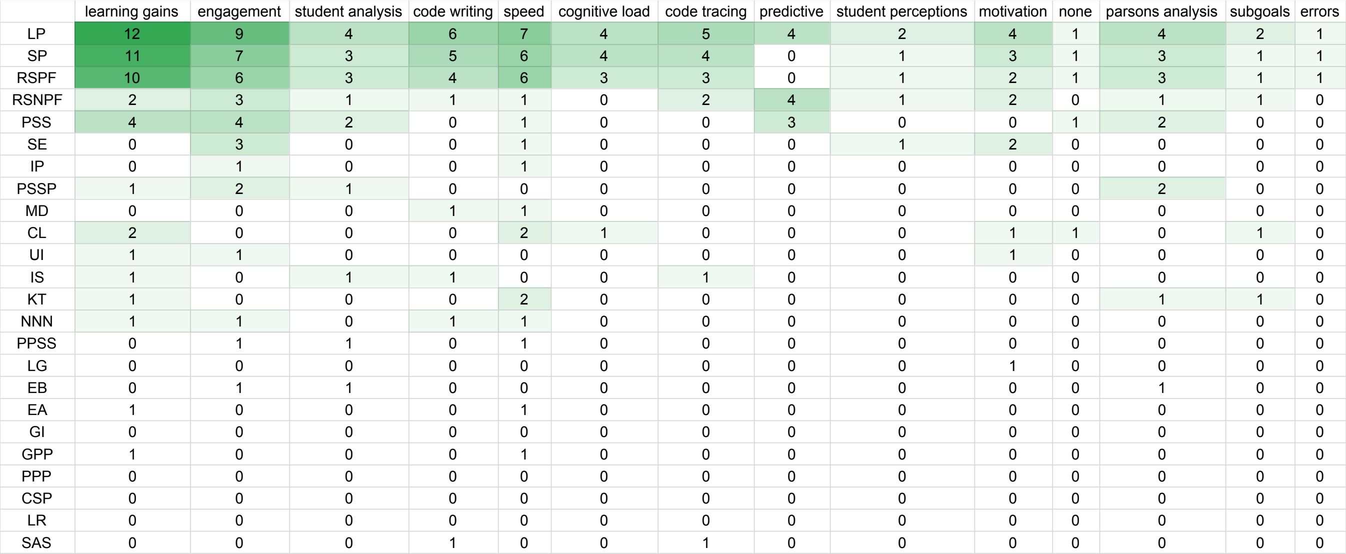
5.1 Contextual Research Gaps

A large proportion of the identified studies on Parsons problems focused on CS1- and CS2-level courses in tertiary education, covering elementary concepts such as loops, conditionals, variables, lists and arrays, and functions. At the present state, although there is plenty of evidence of Parsons problems supporting learning a new task and topic, the use of Parsons problems in more advanced courses and more advanced topics is less common. There is a clear research gap that calls for researchers to study whether and to what extent learners would benefit from the use of Parsons problems in advanced algorithm-focused classes such as Design and Analysis of Algorithms, Randomized Algorithms, and Computational Geometry. Further, the benefits of Parsons problems in software engineering related classes such as Web Software Development should also be explored — the latter course in particular could provide insight into the benefits of Parsons problems when working with applications that consist of multiple files and multiple types of code (e.g. SQL, JavaScript, HTML, CSS).

When looking into the number of participants in the studies, we observed a good scatter of participant counts with a median around 102. As the majority of the studies focused on a single context, there was little discussion on the effect of the class size on the benefits of the Parsons problems. We see a clear need in studying the benefits of Parsons problems in classes of different sizes. One could, as an example, hypothesize that students in smaller classes could have more direct support available from the course instructors, which could in a sense influence the observed benefits of Parsons problems. In the same vein, the benefits of Parsons problems could be more considerable in larger classes with less opportunities of support from instructors. Of course, when conducting such studies, the contextual factors such as available support should be made more explicitly clear.

As we further reviewed the studies in terms of contextual variables and sample sizes, we observed less-than-expected consideration of the prior experiences of the participants when reporting and reflecting on the study results. Given that Parsons problems are often used as a supporting tool for learning programming, one should look into the benefits of Parsons problems for students with different backgrounds and identities. As an example, further studies should look into to what extent students' prior programming background influences the utility of Parsons problems, and also into at what point can Parsons problems become harmful (e.g. due to the expertise reversal effect [209]).

Further, when considering the variety of learning management systems and tools used for delivering Parsons problems, there is a clear need to understand the affordances of these tools and systems and consequently their effect on the use of Parsons problems and learning. Given that diverse systems deliver Parsons problems, is it possible that some study results could be explained by the tools used



	learning gains	engagement	student analysis	code writing	speed	cognitive load	code tracing	predictive	student perceptions	motivation	none	parsons analysis	subgoals	errors
LP	12	9	4	6	7	4	5	4	2	4	1	4	2	1
SP	11	7	3	5	6	4	4	0	1	3	1	3	1	1
RSPF	10	6	3	4	6	3	3	0	1	2	1	3	1	1
RSNPF	2	3	1	1	1	0	2	4	1	2	0	1	1	0
PSS	4	4	2	0	1	0	0	3	0	0	1	2	0	0
SE	0	3	0	0	1	0	0	0	1	2	0	0	0	0
IP	0	1	0	0	1	0	0	0	0	0	0	0	0	0
PSSP	1	2	1	0	0	0	0	0	0	0	0	2	0	0
MD	0	0	0	1	1	0	0	0	0	0	0	0	0	0
CL	2	0	0	0	2	1	0	0	0	1	1	0	1	0
UI	1	1	0	0	0	0	0	0	0	1	0	0	0	0
IS	1	0	1	1	0	0	1	0	0	0	0	0	0	0
KT	1	0	0	0	2	0	0	0	0	0	0	1	1	0
NNN	1	1	0	1	1	0	0	0	0	0	0	0	0	0
PPSS	0	1	1	0	1	0	0	0	0	0	0	0	0	0
LG	0	0	0	0	0	0	0	0	0	1	0	0	0	0
EB	0	1	1	0	0	0	0	0	0	0	0	1	0	0
EA	1	0	0	0	1	0	0	0	0	0	0	0	0	0
GI	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GPP	1	0	0	0	1	0	0	0	0	0	0	0	0	0
PPP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CSP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LR	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SAS	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Figure 12: Cross tabulation between the thematic research areas (RQ2) and the research evidence (RQ3) presented in Parsons related papers showing what type of evidence has been presented for the different research themes. Please note that a paper could be tagged with multiple different themes and/or types of research evidence, and that only papers tagged with IC1 (empirical papers) are included.

in delivering the problems? If so, the results may not generalize to cases using alternate tools.

Students learning to use the systems should also be considered more often; as an example, rather few research studies on Parsons problems utilized practice sessions where students would first learn to use the system, which could lead to confounding issues that relate to the usability of such systems. We acknowledge that the issue is also present in other studies that deliver content using a system.

Related to usability is the lack of research on Parsons problems and error messages. Programming error messages (PEMs) are widely known to be a source of difficulty for novices [14]. Recent research has attempted to close this gap by attempting to understand how to make more usable error messages for novice learners [15, 16, 44, 46, 186, 187], but this usually centers around compiler or other syntax error messages reported at compile or runtime. One benefit to using Parsons problems is that we have the opportunity to rethink PEMs in this context. Since the problem domain has been drastically reduced, so too have the errors that can and should be shown to students. This also means that new types of errors are possible that would not be made by students writing code in an IDE. It’s possible that existing guidelines for error messages could be useful [14], though this has not been empirically confirmed. It’s also possible that information on novices moving to a new programming language could be useful [42] given the stark differences between programming from scratch and using Parsons problems. We call for more research on this topic.

Finally, naturally, when new types of Parsons problems are introduced, their effect should be compared and contrasted with other types of Parsons problems, and preferably studied in more than one context.

5.2 Thematic Research Gaps

When considering the research themes discussed in Section 4.3, we observe that many of the themes have occurred rarely altogether and that, regarding jointly reoccurring research themes, there are various themes that have not appeared together. In addition to the empty areas in the heatmap in Figure 9, which reveal research gaps, the areas with only a few papers could require further attention. As an example, as also already discussed above, the impact of the systems used on students’ learning should require further attention, as presently none of the articles explicitly study the user interfaces (UI) jointly with cognitive load (CL) or student engagement (SE). Given that the user interface can be a considerable confounding factor and influence both cognitive load and engagement, variations in user interfaces should be explored — as an example, one could study whether having to sort fragments in place or having to move and sort fragments to a specific solution area would yield different levels of cognitive load and whether it would affect student engagement. Naturally, this would have to take into account the specific types of Parsons problems, such as using distractors in an area where the only possibility is to sort the fragments would likely lead to broken programs, if there was no functionality to comment code or otherwise remove the distractors.

Further, when considering cognitive load, it is perhaps surprising that identifying behavior that could lead to increased cognitive load is also relatively rarely explored. As an example, using Problem Solving Solution Paths (PSSP), one could identify states in the problem solving process that are likely related to higher cognitive load, and also look into what pathways lead to those states. For students who are on such a path, PSSP analysis could inform UI design, provide early feedback or interventive scaffolding (IS), and assess the effectiveness of such feedback on students’ learning.

Interestingly, also, the Problem Solving Solution Paths (PSSP) seem to not have been studied with Skill Acquisition Sequence (SAS), although how a student solves an individual Parsons problem could likely provide insights into their skills and, when using multiple Parsons problems, and their possible sequencing.

Very few studies have been done that relate to identity issues for various underrepresented learner groups. We identified only three papers that consider the intersection of gender identity (GI) and Parsons problems, and we found none considering the intersections with many other under-served identity groups.

Lastly, we found that papers related to mobile devices (MD) had not considered instructor perspectives (IP) or predicting student success (PSS), which would make novel future directions for research.

5.3 Gaps in Research Evidence

While there is strong evidence for several benefits from Parsons problems, there is a clear need for replication studies and more multi-institutional and multi-national studies. Most of the research studies were conducted at a single institution with less than 500 subjects. Some proposed benefits from using Parsons problems, such as using them to predict struggling students or to understand student errors, had a small number of participants. Since much of the current research was done with introductory concepts in Python and Java, there is a clear need for more studies using other languages and/or more advanced concepts. Also, most of the current studies were conducted in classrooms, so more work should be done in online and hybrid formats. In addition, more research studies are needed on some of the newer types of Parsons problems such as adaptive, faded, or mobile.

Only one study compared the learning gain from solving Parsons problems with distractors versus those without. This study claimed that distractors hurt learning efficiency [95]. However, it only studied one type of distractor (suboptimal path) and only assessed learning on Parsons problems without any distractors. Parsons and Haden hypothesized that distractors help students learn to recognize and fix common syntax errors while writing code [177]. A qualitative study provided evidence that teachers who were learning to program felt that solving Parsons problems with distractors helped them learn to fix and write code [57]. Morin and Kecskemety reported that 60% of their undergraduate students were positive about distractors [162]. More research needs to be done with and without distractors to clarify their effect on learning and student engagement.

In addition, only one study compared solving Parsons problems with execution-based feedback versus line-based feedback [100]. That study found benefits and issues with both types. However, it did not compare the two types with respect to learning gains. Students who needed more than one attempt to solve the problem took longer to reach a correct solution with execution-based feedback than with line-based feedback, while students with execution-based feedback requested feedback less frequently. Regarding benefits and issues, execution-based feedback can allow multiple possible solutions, while line-based systems typically have only one correct solution. Similarly, line-based feedback highlights problem areas better than execution-based feedback. Some Parsons systems have used execution-based feedback [84, 96, 225] while others have used line-based feedback [59, 109, 128]. More research should be done to determine the effects of different types of feedback.

While there is evidence that most students find Parsons problems engaging [161, 177], there is also evidence that some students strongly dislike them [98]. One study found evidence that Parsons problems were beneficial for novice students, but that fix and write code activities were more beneficial for more advanced students [69]. More work should be done to determine who benefits from solving Parsons problems and under what conditions. Since Parsons problems are a type of scaffolding, work should also be done to determine how to best fade that scaffolding as expertise develops.

Only two randomized controlled studies have been done comparing learning gains from solving Parsons problems versus writing the equivalent code [59, 62]. These were both conducted with undergraduate students in Python. Most computer science courses require students to mostly practice by writing textual code from scratch. More evidence is needed to convince instructors that Parsons problems can help students learn to fix and/or write code.

6 PARSONS STUDY IN A BOX

There have been few multi-institutional and multinational studies of Parsons problems [69–71], and those that have been conducted have had fewer than 100 participants. Large scale multi-institutional and multinational studies are needed to provide generalized and robust evidence of the effectiveness of Parsons problems, and to reveal different effects by context. We designed and pilot-tested materials for several studies based on the identified need for research in three areas related to learning. Specifically, we focused on comparing learning between the following pairs of conditions:

(1) solving adaptive Parsons problems versus writing the equivalent code
(2) solving adaptive Parsons problems with and without distractors
(3) using a Parsons problem to scaffold students while writing code versus no scaffolding

Since most of the research studies to date have been on basic introductory concepts like loops and conditionals, two of the studies focus on writing classes in Python. For deploying the studies, we selected the Runestone Academy platform because it supports adaptive Parsons problems [59], is a robust platform already used by thousands of students, and logs all interactions. Our goal was to create 'studies-in-a-box' that include all the information and resources that an instructor would need to contribute to one or more multi-national and multi-institutional studies on the effectiveness of Parsons problems.

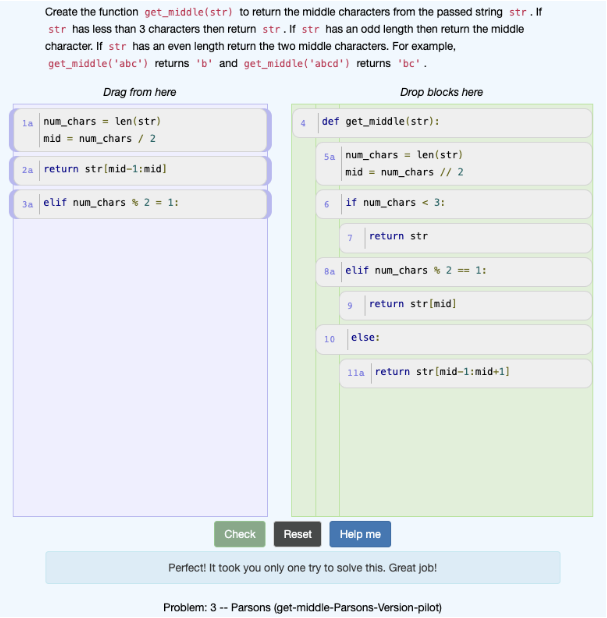
It is important to note that there can be accessibility issues in Parsons problems as they are typically visual and require the user to drag and drop blocks. Runestone Academy added the ability to use a tab to move blocks for students with accessibility issues. In addition, Runestone is in the process of changing authoring languages from reStructuredText to PreText in part because unlike most other markup languages, PreText is designed to serve documents in a wide variety of output formats: PDF, HTML, EPUB, Jupyter notebooks, and electronic or embossed braille. Special attention has been given in PreText to making the HTML output as accessible as possible such as the inclusions of tactile outputs for both mathematical and graphical output.

6.1 Development Process

We first gathered information in a Google form from the working group members about opportunities for running a study in the summer and fall. This form included the programming language, number of students, location, type of opportunity (summer camp, undergraduate course, etc), and topics covered. Based on these opportunities, we created materials for an experiment on the basics of Python 3. We leveraged practice problems that had been used in other research [98]. Participants were randomly assigned to one of two types of practice: Parsons problems or write code problems. See Figure 13 for an example Parsons problem and Figure 14 for the equivalent write code problem.

Since students in this study were not familiar with the Runestone platform, we created a page to introduce the types of problems. The page has videos that explain how to solve adaptive Parsons problems and write code problems. It also had very simple practice problems to verify that participants can solve each problem type.

Since we did not have an opportunity to test the materials with students we added a feedback box to the end of each page in the study. We revised the materials based on this feedback.



Create the function `get_middle(str)` to return the middle characters from the passed string `str` . If `str` has less than 3 characters then return `str` . If `str` has an odd length then return the middle character. If `str` has an even length return the two middle characters. For example, `get_middle('abc')` returns `'b'` and `get_middle('abcd')` returns `'bc'` .

Drag from here

```
1a  num_chars = len(str)
    mid = num_chars / 2
2a  return str[mid-1:mid]
3a  elif num_chars % 2 = 1:
```

Drop blocks here

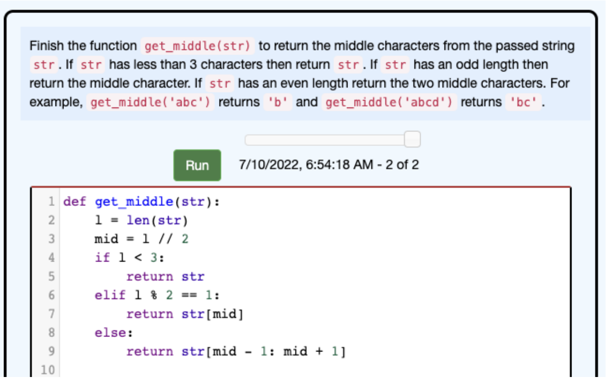
```
4   def get_middle(str):
5a      num_chars = len(str)
        mid = num_chars // 2
6       if num_chars < 3:
7           return str
8a      elif num_chars % 2 == 1:
9           return str[mid]
10      else:
11a         return str[mid-1:mid+1]
```

Perfect! It took you only one try to solve this. Great job!

Problem: 3 -- Parsons (get-middle-Parsons-Version-pilot)

Figure 13: Example question shown to participants in the practice condition that used Parsons problems

There were five problems in each of the practice conditions and five problems on the post-test. Three of the post-test problems were write code problems with unit tests as shown in Figure 15 and two were fix code problems with errors like the distractors in the Parsons problem that also had unit tests. The post-test problems were scored based on the percentage of unit tests that pass. In order to keep the activity length for this initial pilot short, we did not include a separate set of questions for a pre-test.

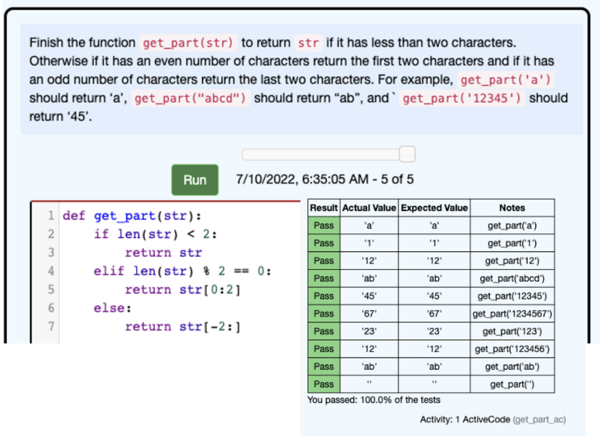


Finish the function `get_middle(str)` to return the middle characters from the passed string `str` . If `str` has less than 3 characters then return `str` . If `str` has an odd length then return the middle character. If `str` has an even length return the two middle characters. For example, `get_middle('abc')` returns `'b'` and `get_middle('abcd')` returns `'bc'` .

7/10/2022, 6:54:18 AM - 2 of 2

```
def get_middle(str):
    l = len(str)
    mid = l // 2
    if l < 3:
        return str
    elif l % 2 == 1:
        return str[mid]
    else:
        return str[mid - 1: mid + 1]
```

Figure 14: Example question shown to participants in the practice condition that used write code problems



Finish the function `get_part(str)` to return `str` if it has less than two characters. Otherwise if it has an even number of characters return the first two characters and if it has an odd number of characters return the last two characters. For example, `get_part('a')` should return 'a', `get_part("abcd")` should return "ab", and ` `get_part('12345')` should return '45'.

7/10/2022, 6:35:05 AM - 5 of 5

```
def get_part(str):
    if len(str) < 2:
        return str
    elif len(str) % 2 == 0:
        return str[0:2]
    else:
        return str[-2:]
```

Result	Actual Value	Expected Value	Notes
Pass	'a'	'a'	get_part('a')
Pass	'1'	'1'	get_part('1')
Pass	'12'	'12'	get_part('12')
Pass	'ab'	'ab'	get_part('abcd')
Pass	'45'	'45'	get_part('12345')
Pass	'67'	'67'	get_part('1234567')
Pass	'23'	'23'	get_part('123')
Pass	'12'	'12'	get_part('123456')
Pass	'ab'	'ab'	get_part('ab')
Pass	''	''	get_part('')

You passed: 100.0% of the tests

Activity: 1 ActiveCode (get_part_ac)

Figure 15: Example of one of the write code questions on the post-test with unit test results

6.2 Pilot Test

We tested the study materials with novice students learning to program in Python at both DePaul University and the University of Virginia. These students were familiar with the basics of Python 3 including variables, strings, conditionals, functions, list, tuples, and dictionaries.

Instructors asked students to voluntarily participate in the pilot test at DePaul University shortly after the semester had ended. Two students started the practice but did not complete it. One student correctly completed all the Parsons practice problems but did not attempt any of the post-test questions. Instructors also asked students to voluntarily participate in the pilot test at the University of Virginia. Nine students from the University of Virginia correctly completed all of the problems in the familiarization page. Only six (67%) of the students continued to the practice problems. Two (33%) were in the write code condition and four (67 %) were in the Parsons condition. The average score for the write code condition was 36% while the average score for the Parsons condition was 51%. Only one student (11%) completed the post-test with a score of 64%. Very low participation rates in voluntary software evaluations like this are common, as we have earlier noted in this report (for example, see discussion of [25] in Section 2.2.1).

6.3 Changes Based on the Pilot Test

Since very few students voluntarily completed all parts of the pilot test, we recommend that instructors conduct a study as part of regular class activities, such as in a lab or discussion group, and give credit to students who at least attempt every section. This will require that students login with an identifier that instructors can use to assign credit. Instructors can generate anonymous logins and passwords for students and keep a map from the generated login to their actual identifier, or we can share a script to run on the log file to anonymize the data before it is shared. Instructions for both of these approaches are included in our study-in-a-box documents.

In our pilot, the difficulty of the practice problems and post-test questions may have discouraged some of the participants from completing the study. We have modified the study since then to cover fewer topics by removing questions on dictionaries. We added text on both the practice and post-test page that says, "Please answer the following problems to the best of your ability without any outside help. You can stop working on a problem after you worked on it for about five minutes without solving it."

6.4 Studies-in-a-box

All materials for these studies, including contact information, can be accessed on our online resource page[6].

All studies start with a page explaining the purpose of the study, the estimated time for the study, and the parts of the study. All studies have an estimated time of 50 minutes, designed to fit into a typical lab/discussion period. There are four parts to the studies: a pre-survey about the student's experience and confidence, an introduction to the different types of problems with videos and practice problems, a set of practice problems with two conditions (A/B), and a set of post-test problems. Learners are randomly placed in either condition A or B.

We also created an optional pre-test and an optional post survey. The pre-test will allow us to establish that the groups have equivalent prior knowledge, and the post survey will permit analyses by gender identity, age, major, and amount of prior experience with programming in Python.

The steps to run a study are as follows:

(1) Seek approval for the study from the local Institutional/Ethics Review Board. On the online resource page, we provide sample IRB applications.
(2) Fill out an online form with basic information about your context including the programming language, type of course or opportunity, number of students, if you are giving any points/credit for completing the study and if so how many.
(3) Create a custom course on Runestone Academy for your students.
(4) You can either ask your students to register for your custom ebook using an institutional identifier such as student name or number or create user names for your students and handle the mapping from user name to the student identifier.
(5) Optionally have students take the pre-test in the ebook.
(6) Run the study. We recommend running the study in a lab / discussion or lecture if possible and giving credit for completing the study.
(7) Optionally have students fill out the post survey.
(8) If you want to see how your students performed you can use the grading interface on the instructor's page to grade the practice and post-test.
(9) Download the log file for analysis from the instructor's page and submit it to a shared online drive for analysis. If the log file contains any student identifiable data we will provide a script to convert it to an anonymized format first or alternatively we can request an anonymized log file for that custom course from Runestone.

We now describe three studies that are ready for deployment. These studies are all for Python 3.

6.5 Study 1: Solving Adaptive Parsons vs Write Code

While several studies have provided evidence that students can solve Parsons problems significantly more quickly than writing the equivalent textual code [59, 62] or assembling the equivalent blocks in a block-based environment [234] with no negative effects on learning, these studies have been conducted at single institutions in a single country. In addition, recent research found that students are not always significantly faster at solving Parsons problems versus writing the equivalent code when the Parsons problem solution does not match the most common student solution and/or when there are many different possible solutions [55, 98]. It is important to test the learning efficiency of solving Parsons problems versus writing the equivalent code in many languages, contexts, institutions, and countries.

This study uses Python 3. It should be conducted after students have been introduced to the covered concepts (variables, modulus, Boolean flags, strings, conditionals, loops, functions, and lists), but have not yet mastered loops and lists.

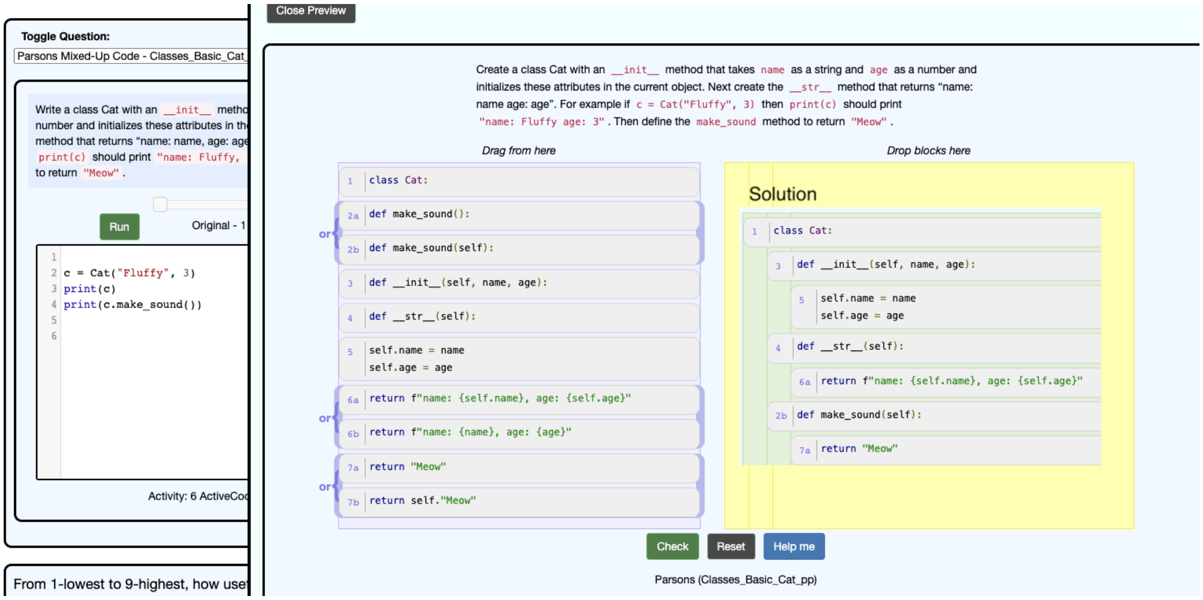
Students will be randomly assigned to one of two practice conditions: adaptive Parsons problems or the equivalent write code problems. The practice problems have been used in other studies [98] and have been revised to ensure that the Parsons solution matches the most common student written solution. The fix code problem in the post-test has errors that match the distractors in the Parsons problems. The study materials are available in a free ebook[7]. If you already have a Runestone account you can add the course name of "p3pt".

6.6 Study 2: Solving Adaptive Parsons Problems with Distractors vs No Distractors

As noted in Section 5.3, only one prior research study compared learning gains from solving Parsons problems with distractors versus without distractors [95]. This work was conducted in a blocks-based environment and only used one type of distractor (suboptimal

[6] https://iticse22-parsons-problems.github.io/

[7] https://tinyurl.com/2p8t6yup



Write a class Cat with an `__init__` metho
number and initializes these attributes in th
method that returns "name: name, age: age
`print(c)` should print `"name: Fluffy,`
to return `"Meow"` .

```
c = Cat("Fluffy", 3)
print(c)
print(c.make_sound())
```

From 1-lowest to 9-highest, how use

Create a class Cat with an `__init__` method that takes `name` as a string and `age` as a number and initializes these attributes in the current object. Next create the `__str__` method that returns "name: name age: age". For example if `c = Cat("Fluffy", 3)` then `print(c)` should print `"name: Fluffy age: 3"` . Then define the `make_sound` method to return `"Meow"` .

Drag from here

```
1  class Cat:
2a def make_sound():
2b def make_sound(self):
3  def __init__(self, name, age):
4  def __str__(self):
5  self.name = name
   self.age = age
6a return f"name: {self.name}, age: {self.age}"
6b return f"name: {name}, age: {age}"
7a return "Meow"
7b return self."Meow"
```

Drop blocks here

Solution

```
1  class Cat:
    3  def __init__(self, name, age):
        5  self.name = name
           self.age = age
    4  def __str__(self):
        6a return f"name: {self.name}, age: {self.age}"
    2b def make_sound(self):
        7a return "Meow"
```

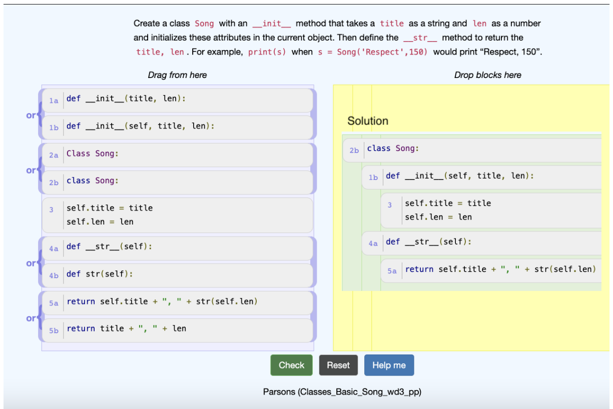
Parsons (Classes_Basic_Cat_pp)

Figure 16: Write code problem with a Parsons problem as scaffolding

path). The post assessment required learners to put only the correct blocks in order. Parsons and Haden hypothesized that solving Parsons problems with distractors would help students learn to recognize and avoid common errors [177]. A qualitative study provided evidence that teachers found solving Parsons problems with distractors useful for learning to fix and write code [57]. However, more work needs to be done to determine if distractors can help students learn to recognize and avoid common errors.

This study uses Python 3. The concepts covered include variables, creating a class, and writing reserved methods (such as `__init__` and `__str__`) and writing new methods. This study is intended to be conducted before learners have started to learn how to write a class. The study materials include a short introduction about how to create a class and methods in Python.

Students will be randomly assigned to one of two practice conditions: solving Parsons problems with distractors or solving Parsons problems without distractors. The study materials are in a free ebook[8]. One of the practice problems with paired distractors is shown in Figure 17. If you already have a Runestone account you can add the course name of "class-exp".



Create 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".

Drag from here

```
1a def __init__(title, len):
1b def __init__(self, title, len):
2a Class Song:
2b class Song:
3  self.title = title
   self.len = len
4a def __str__(self):
4b def str(self):
5a return self.title + ", " + str(self.len)
5b return title + ", " + len
```

Drop blocks here

Solution

```
2b class Song:
    1b def __init__(self, title, len):
        3  self.title = title
           self.len = len
    4a def __str__(self):
        5a return self.title + ", " + str(self.len)
```

Parsons (Classes_Basic_Song_wd3_pp)

Figure 17: An adaptive Parsons problem with paired distractors shown to students in the practice condition

6.7 Study 3: Solving Write Code versus Write Code with Adaptive Parsons as Scaffolding

Many students struggle while writing textual code from scratch [124]. Hou, Ericson, and Wang have recently started using Parsons problems as a type of scaffolding for students who are struggling while writing textual code in Python [106]. They found that students used the Parsons problem to reduce the difficulty of the write code problem, to get a sense for the types of elements in a solution, as a type of directed search, and to help them debug. However, they had a ceiling effect on the pre-test [106], so more work needs to be done to determine the effectiveness of this approach.

This study uses Python 3. The materials in this study are the same as in Study 2 other than the practice problems. This study is intended to be conducted before learners have started to learn how to write a class.

Students will be randomly assigned to one of two practice conditions: solving write code problems without any scaffolding or solving write code problems with Parsons problems as scaffolding as shown in Figure 16. The study materials are in a free ebook[9]. If you already have a Runestone account you can add the course name of "class-tog".

[8] https://tinyurl.com/3j47kt8s

[9] https://tinyurl.com/34jjck96

7 CONCLUSION

Parsons problems have a long history in computing education practice, providing convenient scaffolding for students learning to program. In recent years, both research and classroom interest in Parsons problems have increased due to the widespread adoption of cloud-based educational platforms into which they are directly integrated. This working group has conducted an extensive review of the literature, classifying prior research with respect to study contexts, research foci, and the extent and quality of the presented empirical evidence. As a result, we have been able to highlight current research gaps and thus design targeted study protocols and associated resources which we now invite the computing education research community to utilise.

In this report we have explored the origins of Parsons problems and outlined their defining characteristics. Our systematic review of the literature identified 141 relevant publications which we have analyzed and reported with respect to three primary research questions. Specifically, we investigated the range of contexts in which studies were conducted and the types of Parsons problem features that were used (RQ1). We explored the research questions that guided the work and thus were of interest to the community (RQ2), and we assessed the quality of the evidence that currently exists for the benefits (and limitations) of Parsons problems (RQ3). Finally, we identified existing gaps in the current research literature and have produced experimental protocols and resources that we welcome the research community to use. Our hope is that these resources will enable replication of studies at a large scale and in a variety of contexts, resulting in more generalizable findings.

Through our findings relating to the quality of evidence, we emphasize the need for better reporting standards in computing education research. Researchers should explicitly report:

- study designs, including which variation of the instructional material was used (e.g., faded Parsons problems, with or without distractors, type of feedback),
- definitive participant numbers, including the initial number that began the study as well as how many completed the entire intervention,
- participant background information, especially previous programming experience,
- information on study contexts which should include the institution characteristics, course information, programming language, if and how participants were incentivized to complete the study, etc., and
- specific information about the instructional material used, including any variation on the standard approach.

Finally, our hope is that this working group report will serve as a useful resource for researchers interested in exploring Parsons problems, whether they are new to the area and seeking to understand the current landscape of research, or are experienced researchers looking to identify new and fruitful directions. In order to better understand the key factors that determine the success or failure of Parsons problems in the classroom, there is a need for replication studies in a broad range of contexts. In addition to the research gaps highlighted in this report, as new types of Parsons problems are developed and deployed there remains an ongoing need for their evaluation. We also hope this report can serve educators who are interested in using Parsons problems in the classroom to improve a range of student outcomes, including engagement, satisfaction and learning.

REFERENCES

[1] Kamil Akhuseyinoglu, Jordan Barria-Pineda, Sergey Sosnovsky, Anna-Lena Lamprecht, Julio Guerra, and Peter Brusilovsky. 2020. Exploring Student-Controlled Social Comparison. In *European Conference on Technology Enhanced Learning*. Springer, 244–258.

[2] Laila Al-Malki and Maram Meccawy. 2022. Investigating Students' Performance and Motivation in Computer Programming through a Gamified Recommender System. *Computers in the Schools* (2022), 1–26.

[3] Abdullah Al-Sakkaf, Mazni Omar, and Mazida Ahmad. 2019. Social worked-examples technique to enhance student engagement in program visualization. *Baghdad Science Journal* 16, 2 (2019), 0453.

[4] John Allen and Caitlin Kelleher. 2021. Quantifying Novice Behavior, Experience, and Mental Effort in Code Puzzle Pathways. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–6.

[5] J Thomas Allen, Temi Bidjerano, and Trilby Hren. 2017. What do novices think about when they program? *Journal of Computing Sciences in Colleges* 33, 2 (2017), 171–181.

[6] Bedour Alshaigy. 2017. Evaluation of PILeT: Design guidelines, usability and learning outcomes results. In *2017 IEEE Global Engineering Education Conference (EDUCON)*. 1580–1584. https://doi.org/10.1109/EDUCON.2017.7943059

[7] Bedour Alshaigy, Samia Kamal, Faye Mitchell, Clare Martin, and Arantza Aldea. 2015. Pilet: An interactive learning tool to teach python. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. 76–79.

[8] 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.

[9] Albert Bandura. 1997. *Self-efficacy: The exercise of control*. W.H. Freeman.

[10] Christina Barbieri and Julie L Booth. 2016. Support for struggling students in algebra: Contributions of incorrect worked examples. *Learning and Individual Differences* 48 (2016), 36–44.

[11] ATM Bari, Alessio Gaspar, R Paul Wiegand, Jennifer L Albert, Anthony Bucci, and Amruth N Kumar. 2019. EvoParsons: design, implementation and preliminary evaluation of evolutionary Parsons puzzle. *Genetic Programming and Evolvable Machines* 20, 2 (2019), 213–244.

[12] A.T.M. Golam Bari, Alessio Gaspar, R. Paul Wiegand, Dmytro Vitel, Kok Cheng Tan, and Stephen John Kozakoff. 2019. On the Potential of Evolved Parsons Puzzles to Contribute to Concept Inventories in Computer Programming. In *2019 ASEE Annual Conference & Exposition*. ASEE Conferences, Tampa, Florida. https://peer.asee.org/33142.

[13] Ida Sue Baron. 2005. Test review: Wechsler intelligence scale for children-(WISC-IV). *Child Neuropsychology* 11, 5 (2005), 471–475.

[14] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, , Janice L Pearce, and James Prather. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education* (2019), 177–210.

[15] Brett A Becker, Paul Denny, James Prather, Raymond Pettit, Robert Nix, and Catherine Mooney. 2021. Towards Assessing the Readability of Programming Error Messages. In *Australasian Computing Education Conference*. 181–188.

[16] Brett A Becker, Paul Denny, Janet Siegmund, and Andreas Stefik. 2022. The Human Factors Impact of Programming Error Messages (Dagstuhl Seminar 22052). In *Dagstuhl Reports*, Vol. 12. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[17] Brett A Becker, Catherine Mooney, Amruth N Kumar, and Sean Russell. 2021. A Simple, Language-Independent Approach to Identifying Potentially At-Risk Introductory Programming Students. In *Australasian Computing Education Conference*. 168–175.

[18] Brett A Becker and Keith Quille. 2019. 50 years of cs1 at sigcse: A review of the evolution of introductory programming education research. In *Proceedings of the 50th acm technical symposium on computer science education*. 338–344.

[19] Klara Benda, Amy Bruckman, and Mark Guzdial. 2012. When life and learning do not fit: Challenges of workload and communication in introductory computer science online. *ACM Transactions on Computing Education (TOCE)* 12, 4 (2012), 1–38.

[20] Jeff Bender, Bingpu Zhao, Lalitha Madduri, Alex Dziena, Alex Liebeskind, and Gail Kaiser. 2021. Integrating Parsons Puzzles with Scratch. In *Proceedings of the Twenty-Ninth International Conference on Computers in Education*. 421–431.

[21] Sylvia Beyer, Kristina Rynes, Julie Perrault, Kelly Hay, and Susan Haller. 2003. Gender differences in computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*. 49–53.

[22] Mireilla Bikanga Ada and Mary Ellen Foster. 2021. Enhancing postgraduate students' technical skills: perceptions of modified team-based learning in a six-week multi-subject Bootcamp-style CS course. *Computer Science Education* (2021), 1–25.

[23] William Billingsley and Peter Robinson. 2005. Towards an intelligent online textbook for discrete mathematics. In *Proceedings of the 2005 International Conference on Active Media Technology, 2005.(AMT 2005)*. IEEE, 291–296.

[24] William Billingsley and Peter Robinson. 2007. An Interface for Student Proof Exercises Using MathsTiles and Isabelle/HOL in an Intelligent Book. *Journal of Automated Reasoning* 39, 2 (2007), 181–218.

[25] William Billingsley and Peter Robinson. 2007. Student proof exercises using MathsTiles and Isabelle/HOL in an intelligent book. *Journal of Automated Reasoning* 39, 2 (2007), 181–218.

[26] Alida C Bowler. 1917. A Picture Arrangement Test: Contributed from the Bureau of Juvenile Research, Columbus, Ohio. *The Psychological Clinic* 11, 2 (1917), 37.

[27] John D Bransford, Ann L Brown, and Rodney R Cocking. 2000. *How people learn*. Vol. 11. Washington, DC: National academy press.

[28] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software* 80, 4 (2007), 571–583. https://doi.org/10.1016/j.jss.2006.07.009 Software Performance.

[29] Neil CC Brown and Greg Wilson. 2018. Ten quick tips for teaching programming. *PLoS computational biology* 14, 4 (2018), e1006023.

[30] Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky, et al. 2014. Increasing adoption of smart learning content for computer science education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*. 31–57.

[31] Peter Brusilovsky, Lauri Malmi, Roya Hosseini, Julio Guerra, Teemu Sirkiä, and Kerttu Pollari-Malmi. 2018. An integrated practice system for learning programming in Python: design and evaluation. *Research and practice in technology enhanced learning* 13, 1 (2018), 1–40.

[32] Jeff Carver, Ed Hassler, Elis Hernandes, and Nicholas Kraft. 2013. Identifying Barriers to the Systematic Literature Review Process. *International Symposium on Empirical Software Engineering and Measurement*, 203–212. https://doi.org/10.1109/ESEM.2013.28

[33] Veronica Cateté, Amy Isvik, and Tiffany Barnes. 2020. Infusing computing: A scaffolding and teacher accessibility analysis of computing lessons designed by novices. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–11.

[34] Nick Cheng and Brian Harrington. 2017. The Code Mangler: Evaluating coding ability without writing any code. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 123–128.

[35] Aparna Chirumamilla and Guttorm Sindre. 2019. E-Assessment in Programming Courses: Towards a Digital Ecosystem Supporting Diverse Needs?. In *Proceedings of the IFIP International Federation for Information Processing 2019*. 585–596.

[36] Ruth C Clark, Frank Nguyen, and John Sweller. 2011. *Efficiency in learning: Evidence-based guidelines to manage cognitive load*. John Wiley & Sons.

[37] Graham Cooper and John Sweller. 1987. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of educational psychology* 79, 4 (1987), 347.

[38] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on international computing education research*. 164–172.

[39] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 15 pages. https://doi.org/10.1145/3411764.3445571

[40] Quintin Cutts, Matthew Barr, Mireilla Bikanga Ada, Peter Donaldson, Steve Draper, Jack Parkinson, Jeremy Singer, and Lovisa Sundin. 2019. Experience Report: Thinkathon–Countering an I Got It Working Mentality with Pencil-and-Paper Exercises. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 203–209.

[41] Y. Daniel Liang. 2020. Effective and Innovative Interactives for icseBooks. In *Proceedings of the Future Technologies Conference (FTC) 2019*, Kohei Arai, Rahul Bhatia, and Supriya Kapoor (Eds.). Springer International Publishing, Cham, 890–903.

[42] Paul Denny, Brett A Becker, Nigel Bosch, James Prather, Brent Reeves, and Jacqueline Whalley. 2022. Novice Reflections During the Transition to a New Programming Language. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 948–954.

[43] 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* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[44] Paul Denny, James Prather, and Brett A. Becker. 2020. Error Message Readability and Novice Debugging Performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 480–486. https://doi.org/10.1145/3341525.3387384

[45] 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.

[46] Paul Denny, James Prather, Brett A Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[47] Darina Dicheva and Austin Hodge. 2018. Active learning through game play in a data structures course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 834–839.

[48] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of research on Parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 195–202.

[49] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 21–30.

[50] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Transactions on Computing Education (TOCE)* (2022).

[51] Carol S Dweck. 1986. Motivational processes affecting learning. *American psychologist* 41, 10 (1986), 1040.

[52] Jacquelynne Eccles. 2009. Who am I and what am I going to do with my life? Personal and collective identities as motivators of action. *Educational psychologist* 44, 2 (2009), 78–89.

[53] Elsa Eiriksdottir and Richard Catrambone. 2011. Procedural instructions, principles, and examples: How to structure instructions for procedural tasks to enhance performance, learning, and transfer. *Human factors* 53, 6 (2011), 749–770.

[54] Barbara Ericson. 2019. An Analysis of Interactive Feature Use in Two Ebooks.. In *iTextbooks@ AIED*. 4–17.

[55] Barbara Ericson and Carl Haynes-Magyar. 2022. Adaptive Parsons Problems as Active Learning Activities During Lecture. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. 290–296.

[56] Barbara Ericson, Beryl Hoffman, and Jennifer Rosato. 2020. CSAwesome: AP CSA curriculum and professional development (practical report). In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. 1–6.

[57] 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.

[58] Barbara Ericson, Steven Moore, Briana Morrison, and Mark Guzdial. 2015. Usability and Usage of Interactive Features in an Online Ebook for CS Teachers. In *Proceedings of the WiPSCE '15 Conference*. 111–120.

[59] 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.

[60] 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* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/2787622.2787731

[61] Barbara J Ericson, Hisamitsu Maeda, and Paramveer S Dhillon. 2022. Detecting Struggling Students from Interactive Ebook Data: A Case Study Using CSAwesome. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 418–424.

[62] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Koli Calling 2017,*. 1–10.

[63] 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.

[64] Barbara J Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying design principles for CS teacher Ebooks through design-based research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 191–200.

[65] K Anders Ericsson et al. 2006. The influence of experience and deliberate practice on the development of superior expert performance. *The Cambridge handbook of expertise and expert performance* 38 (2006), 685–705.

[66] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological*

review 100, 3 (1993), 363.

[67] Geela Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2016. Investigating strategies used by novice and expert users to solve Parson's problem in a mobile Python tutor. In *Proc. 9th Workshop on Technology Enhanced Learning by Posing/Solving Problems/Questions*. 434–444.

[68] Geela Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2018. Supporting Novices and Advanced Students in Acquiring Multiple Coding Skills. In *Proceedings of the 26th International Conference on Computers in Education*. 1–6. https://ir.canterbury.ac.nz/handle/10092/16358

[69] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2017. A Comparison of Different Types of Learning Activities in a Mobile Python Tutor. In *Proceedings of the Twenty-Fifth International Conference on Computers in Education*. 604–613.

[70] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2017. Investigating the Effectiveness of Menu-Based Self-explanation Prompts in a Mobile Python Tutor. In *Artificial Intelligence in Education, 18th International Conference, AIED 2017*. 498–501.

[71] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2017. Learning with Engaging Activities via a Mobile Python Tutor. In *Artificial Intelligence in Education*. 613–616.

[72] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2018. Adaptive problem selection in a mobile python tutor. In *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization*. 269–274.

[73] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2019. Evaluation of Parsons problems with menu-based self-explanation prompts in a mobile python tutor. *International Journal of Artificial Intelligence in Education* 29, 4 (2019), 507–535.

[74] Katrina Falkner, Nickolas Falkner, Claudia Szabo, and Rebecca Vivian. 2016. Applying validated pedagogy to MOOCs: An introductory programming course with media computation. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 326–331.

[75] Katia Romero Felizardo, Emilia Mendes, Marcos Kalinowski, Érica Ferreira Souza, and Nandamudi L. Vijaykumar. 2016. Using Forward Snowballing to Update Systematic Reviews in Software Engineering. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 53, 6 pages. https://doi.org/10.1145/2961111.2962630

[76] José Figueiredo and Francisco García-Peñalvo. 2021. A Tool Help for Introductory Programming Courses. In *Ninth International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM'21)* (Barcelona, Spain) *(TEEM'21)*. Association for Computing Machinery, New York, NY, USA, 18–24. https://doi.org/10.1145/3486011.3486413

[77] José Figueiredo and Francisco García-Peñalvo. 2021. Teaching and learning tools for introductory programming in university courses. In *2021 International Symposium on Computers in Education (SIIE)*. IEEE, 1–6.

[78] José Figueiredo, Natália Gomes, and Francisco José García-Peñalvo. 2016. Necourse for learning programming. In *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality*. 549–553.

[79] Jose A. Q. Figueiredo and Francisco Jose Garcia-Penalvo. 2021. Teaching and Learning Strategies for Introductory Programming in University Courses. In *Proceedings of the Ninth International Conference on Technological Ecosystems for Enhancing Multiculturality (TEEM'21)*. 746–751.

[80] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* (2022), 1–29.

[81] Mor Friebroon Yesharim and Mordechai Ben-Ari. 2017. Teaching robotics concepts to elementary school children. In *International Conference on Robotics and Education RiE 2017*. Springer, 77–87.

[82] Rita Garcia. 2021. Evaluating Parsons Problems as a Design-Based Intervention. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.

[83] Rita Garcia, Katrina Falkner, and Rebecca Vivian. 2018. Scaffolding the Design Process Using Parsons Problems. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '18)*. Association for Computing Machinery, New York, NY, USA, Article 26, 2 pages. https://doi.org/10.1145/3279720.3279746

[84] Stuart Garner. 2007. An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Journal of Issues in Informing Science and Information Technology* 4 (2007), 491–501.

[85] Alessio Gaspar, ATM Golam Bari, R Paul Wiegand, Anthony Bucci, Amruth N Kumar, and Jennifer L Albert. 2017. Evolutionary practice problems generation: More design guidelines. In *The Thirtieth International Flairs Conference*.

[86] Alessio Gaspar, Bari Golam, Dmytro Vitel, Paul Wiegand, Jennifer Albert, and Kok Cheng Tan. 2019. Coevolutionary-Aided Teaching: Leveraging the Links Between Coevolutionary and Educational Dynamics. In *Proceedings of the One Hundred Twenty-sixth Annual ASEE Conference and Exposition*. 1–14.

[87] Alessio Gaspar, Dmytro Vitel, and ATM Golam Bari. 2019. Lessons learned from available parsons puzzles software. In *2019 ASEE Annual Conference & Exposition*.

[88] Adam M Gaweda and Collin F Lynch. 2021. Student Practice Sessions Modeled as ICAP Activity Silos. *International Educational Data Mining Society* (2021).

[89] Fernand Gobet and Herbert A Simon. 1996. Recall of random and distorted chess positions: Implications for the theory of expertise. *Memory & cognition* 24, 4 (1996), 493–503.

[90] Olivier Goletti, Kim Mens, and Felienne Hermans. 2021. Tutors' Experiences in Using Explicit Strategies in a Problem-Based Learning Introductory Programming Course. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 157–163.

[91] Shuchi Grover, Roy Pea, and Stephen Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education* 25, 2 (2015), 199–237.

[92] Mark Guzdial and Judy Robertson. 2010. Too much programming too soon? *Commun. ACM* 53, 3 (2010), 10–11.

[93] Simo Haatainen, Antti-Jussi Lakanen, Ville Isomöttönen, and Vesa Lappalainen. 2013. A practice for providing additional support in CS1. In *2013 Learning and Teaching in Computing and Engineering*. IEEE, 178–183.

[94] Kyle J Harms, Evan Balzuweit, Jason Chen, and Caitlin Kelleher. 2016. Learning programming from tutorials and code puzzles: Children's perceptions of value. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 59–67.

[95] Kyle James Harms, Jason Chen, and Caitlin L Kelleher. 2016. Distractors in Parsons problems decrease learning efficiency for young novice programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 241–250.

[96] Kyle J. Harms, Noah Rowlett, and Caitlin Kelleher. 2015. Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles. In *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 271–279.

[97] Brian Harrington and Nick Cheng. 2018. Tracing vs. Writing Code: Beyond the Learning Hierarchy. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 423–428. https://doi.org/10.1145/3159450.3159530

[98] 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.

[99] Sarah Heckman, Jeffrey C. Carver, Mark Sherriff, and Ahmed Al-zubidy. 2021. A Systematic Literature Review of Empiricism and Norms of Reporting in Computing Education Research Literature. *ACM Trans. Comput. Educ.* 22, 1, Article 3 (oct 2021), 46 pages. https://doi.org/10.1145/3470652

[100] Juha Helminen, Petri Ihantola, Ville Karavirta, and Satu Alaoutinen. 2013. How Do Students Solve Parsons Programming Problems? – Execution-Based vs. Line-Based Feedback. In *2013 Learning and Teaching in Computing and Engineering*. 55–61. https://doi.org/10.1109/LaTiCE.2013.26

[101] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How Do Students Solve Parsons Programming Problems? An Analysis of Interaction Traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. Association for Computing Machinery, New York, NY, USA, 119–126. https://doi.org/10.1145/2361276.2361300

[102] Wint Hnin, Michelle Ichinco, and Caitlin Kelleher. 2017. An Exploratory Study of the Usage of Different Educational Resources in an Independent Context. In *Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 181–189.

[103] Amber B Hodgson and N Elizabeth Holland. 2013. Webber® HearBuilder® Sequencing Efficacy Studies: Improved Sequencing Skills for At-Risk Students. (2013). https://www.hearbuilder.com/wp-content/uploads/HearBuilder-Sequencing-Pilot.pdf

[104] Roya Hosseini, Kamil Akhuseyinoglu, Peter Brusilovsky, Lauri Malmi, Kerttu Pollari-Malmi, Christian Schunn, and Teemu Sirkia. 2020. Improving Engagement in Program Construction Examples for Learning Python Programming. *International Journal of Artificial Intelligence in Education* 30 (2020), 299–336.

[105] Roya Hosseini, Kamil Akhuseyinoglu, Andrew Petersen, Christian D Schunn, and Peter Brusilovsky. 2018. PCEX: interactive program construction examples for learning programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. 1–9.

[106] 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 V. 1*. 15–26.

[107] Petri Ihantola, Juha Helminen, and Ville Karavirta. 2013. How to study programming on mobile touch devices: interactive Python code exercises. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. 51–58.

[108] Petri Ihantola and Ville Karavirta. 2010. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. 302–302.

[109] Petri Ihantola and Ville Karavirta. 2011. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education. Innovations in Practice* 10 (2011), 119.

[110] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports* (Vilnius, Lithuania) *(ITICSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[111] Amy Isvik, Veronica Cateté, and Tiffany Barnes. 2021. Investigating the impact of computing vs pedagogy experience in novices creation of computing-infused curricula. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 255–261.

[112] Hiroki Ito, Hiromitsu Shimakawa, and Fumiko Harada. 2020. Advanced Comprehension Analysis Using Code Puzzle Considering the Programming Thinking Ability. In *Special Sessions in the Information Systems and Technologies Track of the Conference on Computer Science and Information Systems*. 45–64.

[113] Hiroki Ito, Hiromitsu Shimakawa, and Fumiko Harada. 2020. Comprehension analysis considering programming thinking ability using code puzzle. In *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems, M. Ganzha, L. Maciaszek, M. Paprzycki (eds). ACSIS*. 609–618. https://doi.org/10.15439/2020F44

[114] Hiroki Ito, Hiromitsu Shimakawa, and Fumiko Harada. 2021. Estimating Learner's Perspective in Programming: Analysis of Operation Time Series in Code Puzzles. In *2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. IEEE, 655–661.

[115] Hiroki Ito, Hiromitsu Shimakawa, and Fumiko Harada. 2022. Process-Oriented Understanding Estimation Using Code Puzzles. *Creative Education* 13, 3 (2022), 750–767.

[116] Erkki Kaila, Einari Kurvinen, Erno Lokkila, and Mikko-Jussi Laakso. 2016. Redesigning an object-oriented programming course. *ACM Transactions on Computing Education (TOCE)* 16, 4 (2016), 1–21.

[117] Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso, Rolf Linden, Einari Kurvinen, Ville Karavirta, and Tapio Salakoski. 2015. Comparing student performance between traditional and technologically enhanced programming course. In *Proceedings of the 17th Australasian Conference on Computing Education* (Sydney, Australia) *(ACE '15)*. Australian Computer Society, Inc., 147–154.

[118] Ville Karavirta, Riku Haavisto, Erkki Kaila, Mikko-Jussi Laakso, Teemu Rajala, and Tapio Salakoski. 2015. Interactive learning content for introductory computer science course using the ville exercise framework. In *2015 International Conference on Learning and Teaching in Computing and Engineering*. IEEE, 9–16.

[119] Ville Karavirta, Juha Helminen, and Petri Ihantola. 2012. A mobile learning application for parsons problems with automatic feedback. In *Proceedings of the 12th koli calling international conference on computing education research*. 11–18.

[120] Sandra Katz, David Allbritton, John Aronis, Christine Wilson, and Mary Lou Soffa. 2006. Gender, achievement, and persistence in an undergraduate computer science program. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* 37, 4 (2006), 42–57.

[121] Elizabeth Kazakoff and Marina Bers. 2012. Programming in a robotics context in the kindergarten classroom: The impact on sequencing skills. *Journal of Educational Multimedia and Hypermedia* 21, 4 (2012), 371–391.

[122] Elizabeth R Kazakoff and Marina Umaschi Bers. 2014. Put your robot in, put your robot out: Sequencing through programming robots in early childhood. *Journal of Educational Computing Research* 50, 4 (2014), 553–573.

[123] Caitlin Kelleher and Wint Hnin. 2019. Predicting cognitive load in future code puzzles. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[124] Paivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: the emotional toll. In *Proceedings of the Sixth international workshop on Computing education research*. 77–86.

[125] Barbara Kitchenham and Stuart Charters. 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering, version 2.3.

[126] Ari Korhonen, Thomas Naps, Charles Boisvert, Pilu Crescenzi, Ville Karavirta, Linda Mannila, Bradley Miller, Briana Morrison, Susan H Rodger, Rocky Ross, et al. 2013. Requirements and design strategies for open source interactive computer science ebooks. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports*. 53–72.

[127] Amruth N Kumar. 2017. The effect of providing motivational support in parsons puzzle tutors. In *International Conference on Artificial Intelligence in Education*. Springer, 528–531.

[128] Amruth N Kumar. 2018. Epplets: a tool for solving parsons puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 527–532.

[129] Amruth N Kumar. 2019. Helping students solve Parsons puzzles better. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 65–70.

[130] Amruth N Kumar. 2019. Mnemonic variable names in Parsons puzzles. In *Proceedings of the ACM Conference on Global Computing Education*. 120–126.

[131] Amruth N. Kumar. 2019. Representing and Evaluating Strategies for Solving Parsons Puzzles. In *Intelligent Tutoring Systems, 15th International Conference, ITS 2019*. 193–203.

[132] Amruth N. Kumar. 2021. Do Students Use Semantics When Solving Parsons Puzzles? – A Log-Based Investigation. In *Intelligent Tutoring Systems*. 444–450.

[133] Amruth N Kumar. 2021. Using Markov Transition Matrix to Analyze Parsons Puzzle Solutions. In *Proceedings of the Workshops at the 14th International Conference on Educational Data Mining*.

[134] Rina PY Lai. 2021. Beyond Programming: A Computer-Based Assessment of Computational Thinking Competency. *ACM Transactions on Computing Education (TOCE)* 22, 2 (2021), 1–27.

[135] Patricia Lasserre and Steven Smithbower. 2010. A proposal for a new communication medium in the classroom. In *Proceedings of the 15th Western Canadian Conference on Computing Education*. 1–5.

[136] Celine Latulipe, N Bruce Long, and Carlos E Seminario. 2015. Structuring flipped classes with lightweight teams and gamification. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 392–397.

[137] Jo-Anne LeFevre and Peter Dixon. 1986. Do written instructions need examples? *Cognition and Instruction* 3, 1 (1986), 1–30.

[138] Raymond Lister, Tony Clear, Dennis J Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, et al. 2010. Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin* 41, 4 (2010), 156–173.

[139] Erno Lokkila, Erkki Kaila, Rolf Lindén, Mikko-Jussi Laakso, and Erkki Sutinen. 2017. Refactoring a CS0 course for engineering students to use active learning. *Interactive Technology and Smart Education* (2017).

[140] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* (dec 2022). https://doi.org/10.1145/3487050

[141] Mike Lopez, Ken Sutton, and Tony Clear. 2009. Surely we must learn to read before we learn to write!. In *Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95*. 165–170.

[142] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[143] Douglas Lusa Krug, Edtwuan Bowman, Taylor Barnett, Lori Pollock, and David Shepherd. 2021. Code Beats: A Virtual Camp for Middle Schoolers Coding Hip Hop. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 397–403.

[144] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[145] Nicholas Lytle, Yihuan Dong, Veronica Cateté, Alex Milliken, Amy Isvik, and Tiffany Barnes. 2019. Position: Scaffolded coding activities afforded by block-based environments. In *2019 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 5–7.

[146] Stephen MacNeil, Celine Latulipe, and Aman Yadav. 2015. Learning in Distributed Low-Stakes Teams. In *Proceedings of the Eleventh Annual International Computing Education Research conference (ICER)*. 227–236.

[147] Hisamitsu Maeda, Barbara Ericson, and Paramveer Dhillon. 2021. Comparing Ebook Student Interactions With Test Scores: A Case Study Using CSAwesome. (2021).

[148] Salil Maharjan and Amruth Kumar. 2020. Analyzing Parsons Puzzle Solutions using Modified Levenshtein's Algorithm.. In *CSEDM@ EDM*.

[149] Salil Maharjan and Amruth N Kumar. 2020. Using Edit Distance Trails to Analyze Path Solutions of Parsons Puzzles. (2020).

[150] Sohail Iqbal Malik, Roy Mathew, Abir Al-Sideiri, Jasiya Jabbar, Rim Al-Nuaimi, and Ragad M. Tawafak. 2022. Enhancing problem-solving skills of novice programmers in an introductory programming course. *Computer Applications in Engineering Education* 30, 1 (2022), 174–194. https://doi.org/10.1002/cae.22450 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.22450

[151] Yana Malysheva and Caitlin Kelleher. 2020. Bugs as Features: Describing Patterns in Student Code through a Classification of Bugs. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[152] Hamza Manzoor, Kamil Akhuseyinoglu, Jackson Wonderly, Peter Brusilovsky, and Clifford A Shaffer. 2019. Crossing the borders: Re-use of smart learning objects in advanced content access systems. *Future Internet* 11, 7 (2019), 160.

[153] Jane Margolis. 2017. *Stuck in the Shallow End, updated edition: Education, Race, and Computing.* MIT press.

[154] Jane Margolis and Allan Fisher. 2002. *Unlocking the clubhouse: Women in computing.* MIT press.

[155] Lauren Margulieux, Paul Denny, Kathryn Cunningham, Michael Deutsch, and Benjamin R. Shapiro. 2021. When Wrong is Right: The Instructional Power of Multiple Conceptions. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/3446871.3469750

[156] Aditya Mehrotra, Christian Giang, Noé Duruz, Julien Dedelley, Andrea Mussati, Melissa Skweres, and Francesco Mondada. 2020. Introducing a paper-based programming language for computing education in classrooms. In *Proceedings of the 2020 ACM conference on innovation and technology in computer science education.* 180–186.

[157] Gregor Milicic, Sina Wetzel, and Matthias Ludwig. 2020. Generic tasks for algorithms. *Future Internet* 12, 9 (2020), 152.

[158] Bradley N Miller and David L Ranum. 2012. Beyond PDF and ePub: toward an interactive textbook. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education.* 150–155.

[159] George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.

[160] Claudio Mirolo. 2010. Learning (through) Recursion: A Multidimensional Analysis of the Competences Achieved by CS1 Students. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (Bilkent, Ankara, Turkey) *(ITiCSE '10)*. Association for Computing Machinery, New York, NY, USA, 160–164. https://doi.org/10.1145/1822090.1822136

[161] Brooke Morin and Krista M. Kecskemety. 2021. Collaborative Parsons Problems in a Remote-learning First-year Engineering Classroom. In *2021 ASEE Virtual Annual Conference Content Access*. ASEE Conferences, Virtual Conference. https://peer.asee.org/36809.

[162] Brooke C. Morin, Krista M. Kecskemety, Kathleen A. Harper, and Paul Alan Clingan. 2020. Work in Progress: Parsons Problems as a Tool in the First-Year Engineering Classroom. In *2020 ASEE Virtual Annual Conference Content Access.* ASEE Conferences, Virtual On line. https://peer.asee.org/35675.

[163] Briana B Morrison, Adrienne Decker, and Lauren E Margulieux. 2016. Learning loops: A replication study illuminates impact of HS courses. In *Proceedings of the 2016 ACM conference on international computing education research.* 221–230.

[164] Briana B Morrison, Lauren E Margulieux, and Adrienne Decker. 2020. The curious case of loops. *Computer Science Education* 30, 2 (2020), 127–154.

[165] 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.

[166] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research.* 21–29.

[167] Yasuichi Nakayama, Yasushi Kuno, and Hiroyasu Kakuda. 2020. Split-Paper Testing: A Novel Approach to Evaluate Programming Performance. *Journal of Information Processing* 28 (2020), 733–743.

[168] Solomon Sunday Oyelere, Friday Josepth Agbo, Ismaila Sanusi, Abdullahi Abubakar Yunusa, and Kissinger Sunday. 2019. Impact of Puzzle-based Learning Technique for Programming Education in Nigeria Context. In *Proceedings of the 2019 IEEE 19th International Conference on Advanced learning Technologies (ICALT).* 239–241.

[169] Solomon Sunday Oyelere, Jarkko Suhonen, and Teemu H Laine. 2017. Integrating parson's programming puzzles into a game-based mobile learning application. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research.* 158–162.

[170] Solomon Sunday Oyelere, Jarkko Suhonen, and Erkki Sutinen. 2016. M-learning: A new paradigm of learning ICT in Nigeria. *International Journal of Interactive Mobile Technologies* 10, 1 (2016).

[171] Solomon Sunday Oyelere, Jarkko Suhonen, Greg M. Wajiga, and Erkki Sutinen. 2018. Design, development, and evaluation of a mobile learning application for computing education. *Education and Information Technologies* 23, 1 (2018), 467–495. https://doi.org/10.1145/3487050

[172] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive load theory and instructional design: Recent developments. *Educational psychologist* 38, 1 (2003), 1–4.

[173] Fred Paas, Tamara Van Gog, and John Sweller. 2010. Cognitive load theory: New conceptualizations, specifications, and integrated research perspectives. *Educational psychology review* 22, 2 (2010), 115–121.

[174] Jennifer Parham-Mocello, Martin Erwig, and Margaret Niess. 2021. Teaching CS Middle School Camps in a Virtual World. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 1–4.

[175] Thomas H. Park, Meen Chul Kim, Sukrit Chhabra, Brian Lee, and Andrea Forte. 2016. Reading Hierarchies in Code: Assessment of a Basic Computational Skill. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 302–307. https://doi.org/10.1145/2899415.2899435

[176] Miranda C Parker, Kantwon Rogers, Barbara J Ericson, and Mark Guzdial. 2017. Students and teachers use an online ap cs principles ebook differently: Teacher behavior consistent with expert learners. In *Proceedings of the 2017 ACM Conference on International Computing Education Research.* 101–109.

[177] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: A fun and effective learning tool for first programming courses. In *Proceedings of the Eighth Australasian Computing Education Conference*, Vol. 52. 157–163.

[178] Dale Parsons, Krissi Wood, and Patricia Haden. 2015. What are we doing when we assess programming. In *Proceedings of the 17th Australasian Computing Education Conference (ACE 2015)*, Vol. 27. 30.

[179] Peter L Pirolli and John R Anderson. 1985. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie* 39, 2 (1985), 240.

[180] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2021. Evaluating Proof Blocks Problems as Exam Questions. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/3446871.3469741

[181] Seth Poulsen, Mahesh Viswanathan, Geoffrey L. Herman, and Matthew West. 2022. Proof Blocks: Autogradable Scaffolding Activities for Learning to Write Proofs. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 428–434. https://doi.org/10.1145/3502718.3524774

[182] 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.

[183] James Prather, John Homer, Paul Denny, Brett Becker, John Marsden, and Garrett Powell. 2022. Scaffolding Task Planning Using Abstract Parsons Problems. In *Proceedings of the 2022 World Conference on Computers in Education (WCCE '22).* 1–10.

[184] James Prather, Lauren Margulieux, Jacqueline Whalley, Paul Denny, Brent N Reeves, Brett A Becker, Paramvir Singh, Garrett Powell, and Nigel Bosch. 2022. Getting By With Help From My Friends: Group Study in Introductory Programming Understood as Socially Shared Regulation. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1.* 164–176.

[185] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* 531–537.

[186] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research.* 41–50.

[187] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research.* 74–82.

[188] David Pritchard and Troy Vasiga. 2013. CS circles: an in-browser python course for beginners. In *Proceeding of the 44th ACM technical symposium on Computer science education.* 591–596.

[189] Ela Pustulka, Kai Krause, Lucia de Espona, and Andrea Kennel. 2021. SQL Scrolls-A Reusable and Extensible DGBL Experiment. In *Proceedings of the 10th Computer Science Education Research Conference.* 39–48.

[190] Teemu Rajala, Erkki Kaila, Rolf Linden, and Einari Kurvinen. 2016. Automatically assessed electronic exams in programming courses. In *Proceedings of the ACSW '16 Multiconference.* 1–8.

[191] Alexander Renkl. 2005. The worked-out-example principle in multimedia learning. *The Cambridge handbook of multimedia learning* (2005), 229–245.

[192] Alexander Renkl. 2014. Learning from worked examples: How to prepare students for meaningful problem solving. (2014).

[193] S Sahebi and P Brusilovsky. 2018. Student performance prediction by discovering inter-activity relations. In *Proceedings of the 11th International Conference on Educational Data Mining, EDM 2018.*

[194] Kim Guan Saw. 2017. Cognitive load theory and the use of worked examples as an instructional strategy in physics for distance learners: A preliminary study. *Turkish Online Journal of Distance Education* 18, 4 (2017), 142–159.

[195] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. 2010. An introduction to program comprehension for computer science educators. *Proceedings of the 2010 ITiCSE working group reports* (2010),

65–86.

[196] Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teaching computer programming with PRIMM: a sociocultural perspective. *Computer Science Education* 29, 2-3 (2019), 136–176.

[197] Clifford A Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L Naps. 2011. Opendsa: beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on computing education research*. 112–117.

[198] Timothy Shanahan, Kim Callison, Christine Carriere, Nell K Duke, P David Pearson, Christopher Schatschneider, and Joseph Torgesen. 2010. Improving Reading Comprehension in Kindergarten through 3rd Grade: IES Practice Guide. NCEE 2010-4038. *What Works Clearinghouse* (2010).

[199] Amal Shargabi, Syed Ahmad Aljunid, Muthukkaruppanan Annamalai, Shuhaida Mohamed Shuhidan, and Abdullah Mohd Zin. 2015. Tasks that can improve novices' program comprehension. In *2015 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)*. IEEE, 32–37.

[200] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2010. Instructor perspectives of multiple-choice questions in summative assessment for novice programmers. *Computer Science Education* 20, 3 (2010), 229–259.

[201] Guttorm Sindre. 2020. Code Writing vs Code Completion Puzzles: Analyzing Questions in an E-exam. In *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.

[202] Teemu Sirkiä. 2016. Combining Parson's problems with program visualization in CS1 context. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. 155–159.

[203] Teemu Sirkiä and Lassi Haaranen. 2017. Improving online learning activity interoperability with acos server. *Software: Practice and Experience* 47, 11 (2017), 1657–1676.

[204] John A Sloboda, Jane W Davidson, Michael JA Howe, and Derek G Moore. 1996. The role of practice in the development of performing musicians. *British journal of psychology* 87, 2 (1996), 287–309.

[205] Juha Sorva and Otto Seppälä. 2014. based design of the first weeks of CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. 71–80.

[206] Ben Stephenson and Guransh Mangat. 2021. Using a Computer to Score Parsons Problems Answered on Paper. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1069–1075. https://doi.org/10.1145/3408877.3432467

[207] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[208] John Sweller. 2006. The worked example effect and human cognition. *Learning and instruction* (2006).

[209] John Sweller, Paul L Ayres, Slava Kalyuga, and Paul Chandler. 2003. The expertise reversal effect. (2003).

[210] John Sweller and Graham A Cooper. 1985. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and instruction* 2, 1 (1985), 59–89.

[211] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* (2019), 1–32.

[212] Grace Tan and Anne Venables. 2010. Wearing the assessment 'BRACElet'. *Journal of Information Technology Education: Innovations in Practice* 9, 1 (2010), 25–34.

[213] PrairieLearn Team. 2021. PrairieLearn Documentation. https://prairielearn.readthedocs.io/en/latest/

[214] Dirk Tempelaar, Bart Rienties, and Quan Nguyen. 2018. Investigating Learning Strategies in a Dispositional Learning Analytics Context: The Case of Worked Examples. In *Proceedings of the 8th International Conference on Learning Analytics and Knowledge* (Sydney, New South Wales, Australia) *(LAK '18)*. Association for Computing Machinery, New York, NY, USA, 201–205. https://doi.org/10.1145/3170358.3170385

[215] John Gregory Trafton and Brian J Reiser. 1994. *The contributions of studying examples and solving problems to skill acquisition*. Ph. D. Dissertation. Citeseer.

[216] The WeBWorK Project (TWP). 2018. WeBWorK. https://openwebwork.org/

[217] Shin Ueno, Yuuki Terui, Ryuichiro Imamura, Yashushi Kuno, and Hironori Egi. 2021. Analysis of the Answering Processes in Split-Paper Testing to Promote Instruction. In *Proceedings of the 29th International Conference on Computers in Education*. 273–278.

[218] Jeroen JG Van Merriënboer and Marcel BM De Croock. 1992. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research* 8, 3 (1992), 365–394.

[219] Lev Semenovich Vygotsky. 1980. *Mind in society: The development of higher psychological processes*. Harvard university press.

[220] Jane Waite, Andrea Franceschini, Sue Sentance, Mathew Patterson, and James Sharkey. 2021. An online platform for teaching upper secondary school computer science. In *United Kingdom and Ireland Computing Education Research conference*. 1–7.

[221] Jennifer Wang, Hai Hong, Jason Ravitz, and Sepehr Hejazi Moghadam. 2016. Landscape of K-12 computer science education in the US: Perceptions, access, and barriers. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 645–650.

[222] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W Price. 2020. Crescendo: Engaging students to self-paced programming practices. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 859–865.

[223] Mark Ward and John Sweller. 1990. Structuring effective worked examples. *Cognition and instruction* 7, 1 (1990), 1–39.

[224] David Wechsler. 1939. *The measurement of adult intelligence.* Williams & Wilkins Co, 75–103), doi=https://doi.org/10.1037/10020-007.

[225] 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.

[226] Matthew West, Nathan Walters, Mariana Silva, Timothy Bretl, and Craig Zilles. 2021. Integrating Diverse Learning Tools using the PrairieLearn Platform. In *Seventh SPLICE Workshop at SIGCSE*.

[227] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[228] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235 arXiv:https://doi.org/10.1080/08993408.2019.1565235

[229] Iman YeckehZaare, Chloe Aronoff, and Gail Grot. 2022. Retrieval-Based Teaching Incentivizes Spacing and Improves Grades in Computer Science Education. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 892–898. https://doi.org/10.1145/3478431.3499408

[230] Iman YeckehZaare and Paul Resnick. 2019. Speed and Studying: Gendered Pathways to Success. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 693–698. https://doi.org/10.1145/3287324.3287417

[231] Iman YeckehZaare, Paul Resnick, and Barbara Ericson. 2019. A spaced, interleaved retrieval practice tool that is motivating and effective. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 71–79.

[232] Robert Mearns Yerkes. 1921. *Psychological examining in the United States army*. Vol. 15. US Government Printing Office.

[233] Laura Zavala and Benito Mendoza. 2017. Precursor skills to writing code. *Journal of Computing Science in Colleges* 32, 3 (2017), 149–156.

[234] Rui Zhi, Min Chi, Tiffany Barnes, and Thomas W Price. 2019. Evaluating the effectiveness of parsons problems for block-based programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 51–59.

[235] Rui Zhi, Thomas W. Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the Impact of Worked Examples in a Novice Programming Environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 98–104. https://doi.org/10.1145/3287324.3287385

[236] Xinming Zhu and Herbert A Simon. 1987. Learning mathematics from examples and by doing. *Cognition and instruction* 4, 3 (1987), 137–166.

A APPENDIX: EXTRACTION SHEET

Reviewer name

Article title

Inclusion / exclusion criteria Parsons problems is an exercise type used for learning and assessing the ability to construct programs (* programs = intentionally allowing also e.g. ordering higher level plans, while limiting the scope to programming). Parsons problems feature a limited set of blocks that the user needs to move to produce a solution (* limited = not open-ended, * move = intentionally allowing also a type where the blocks are in a single area). There can be extra blocks that are not needed for a solution, and the user may be allowed to enter e.g. variable values into the blocks.

- IC1 - Contains empirical results on the use of Parsons problems / collects data from the use of Parsons problems
- IC2 - Describes a system/tool for presenting/delivering Parsons problems
- IC3 - Describes the use of Parsons problems for teaching
- EC1 - Article is not written in English
- EC2 - Article length is less than or equal to 2 pages
- EC3 - Article is not peer reviewed
- EC4 - Article is a thesis or a dissertation
- EC5 - Parsons problems not discussed (mentioned?) in methodology or results (e.g. citing Parsons but not relevant to gist of the paper)
- EC6 - Not related to Parsons problems

Decision If "Exclude" is chosen, there is no need to upload the paper or provide bibtex information. If "Discuss" or "Include" are chosen, please fill in the remainder of the form. New: Use "Exclude but check when building background" if the paper should be excluded from the lit review but it has components that could be useful in building the background.

- Include
- Exclude
- Discuss
- Exclude but check when building background
- Other:

Article PDF Upload PDF of article

Bibtex entry Use Google Scholar format; e.g. @inproceedings du2020review, title=A review of research on Parsons problems, author=Du, Yuemeng and Luxton-Reilly, Andrew and Denny, Paul, booktitle=Proceedings of the Twenty-Second Australasian Computing Education Conference, pages=195–202, year=2020

Article type A "Lab-based" study is a study that is conducted in a research lab (i.e. a highly controlled environment), while a "Classroom-based" study is a study that's conducted in a normal classroom situation (less controlled). A "System" paper describes a system. An "Experience report" is where the author reflects on their personal experience, but does not involve collection and analysis of data from students or the system (such papers used to be common, but are much less common now).

- Lab-based study
- Classroom-based study
- System paper
- Experience report
- Other:

Study context / level At what educational level is the study conducted?

- Uncontextualized
- Primary school (e.g. elementary school, intermediate school, middle school)
- Secondary school (e.g. high school)
- Tertiary education (e.g. college, university): CS1, CS2
- Tertiary education (e.g. college, university): other courses
- Life-long learning (e.g. MOOCs)
- Not applicable
- Other:

Study context / course topic (or NA) A copy-paste description of context / course topic for the purposes of a more detailed thematic analysis of the sorts of contexts where these studies have been conducted. This information can often be found at the beginning of a Methods section.

Study context / course delivery How is the course typically delivered to students? The main distinction will be whether the course is delivered fully "Online" or not. The distinction between Campus-based and Blended/hybrid might be more subtle, and may not be easy to determine from the information given in the paper (if so, feel free to code as "Unclear"). A campus-based course would be one where all teaching activities are conducted on campus, whereas a blended course would have some teaching components (lectures, labs) delivered online.

- Campus-based course
- Blended/hybrid course
- Online course
- Unclear
- NA
- Other:

Study context / Parsons problem grading Did students receive any marks/grades/credit for participating with the Parsons activities described in the paper?

- Not graded
- Graded
- Unclear
- NA
- Other:

Study context / Parson problem usage (count) How many Parsons problems did students solve as part of the Parsons activities described in the paper?

- One
- Less than five
- Five or more
- Unclear
- NA
- Other:

Study context / Parsons problem role What role do the Parsons problems play with respect to the delivery of the course?

- Part of normal instruction
- Bonus activity
- Extracurricular activity
- Unclear

- NA
- Other:

Study context / Country Where does the study take place?

- USA
- Unclear
- NA
- Other:

Number of participants (if available, enter in other) From how many participants were data collected? (if the information is available, report the number of students who actually participated rather than the total number enrolled in the course).

- Not applicable
- Unclear
- Other:

Two or more groups? Were there two or more groups in the experiment (e.g. a control and a treatment group). In this case, the authors should make some attempt to "compare" some measure between the groups.

- Yes
- No
- Unclear
- NA
- Other:

Motivation of work How do the authors motivate the work? (copy-paste, short description, unclear, or NA). Open-text entry

Research questions What are the explicit research questions / research goals / hypotheses in the article? (copy-paste or NA). Open-text entry

Concepts taught What concepts (e.g. loops etc) are being taught with Parsons problems? (copy-paste, short description, unclear, or NA). Open-text entry

Features What are the features of the used Parsons problems? If there are novel or unusual features in the article that aren't covered by the options, document those in the "Other" field.

- Basic (i.e. draggable code lines)
- Faded (i.e. draggable code lines with parts that need to be filled in)
- Distractors (i.e. code lines that are not needed)
- Adaptive
- Optional (i.e. code lines where you have to pick one or more but not all from a set)
- Unclear
- NA
- Other:

Programming langauge What programming language (if any) is used with Parsons problems?

- Pseudocode
- Java
- Python
- C
- C++
- Not programming language focused
- Other:

Analysis type How does the article evaluate aspects of Parsons problems or their impact?

- Qualitatively
- Quantitatively
- The article does not evaluate aspects of Parsons problems

Contribution What is the contribution / what are the key results of the article? Provide a short summary of the main findings.). Open-text entry

Efficacy Does the article provide a clear measurement of the efficacy/effectiveness of Parsons problems? Although the contribution/results documented in the previous field should include mention of efficacy/effectiveness where relevant, the options below may be helpful when we analyze the data to identify articles that make claims regarding effectiveness. Please record if the article makes a positive claim regarding efficacy (of anything) that can be measured, and whether this is contrasted with other options. If findings are neutral or negative, please state under "Other".

- Yes, but without a comparison to one or more other options
- Yes, and with a comparison to one or more other options
- No
- NA
- Other:

Quality assessment An assessment of the research "quality". For the last question, on threats to validity / limitations: code as "Yes" if there is an explicit (sub)section, "Vague" if they are mentioned as part of some other section (i.e. Discussion, Conclusions), or "No" if they are not mentioned.

- Is there a clearly defined research question/hypothesis? [Yes/No/Vague]
- Is the research process clearly described? [Yes/No/Vague]
- Are the results presented with sufficient detail? [Yes/No/Vague]
- Are threats to validity / limitations addressed in an explicit (sub)section? (code as "vague" if discussed, but not in a separate subsection) [Yes/No/Vague]

Additional notes Open-text entry

B APPENDIX: INTERVIEW WITH DALE PARSONS

We conducted an interview with Dale Parsons on 19th July, 2022. The list of questions and a transcription of this interview is provided below.

Q1 First things first, are they "Parsons Programming Puzzles" or "Parson's Programming Puzzles" (both spellings appear in your original paper, although the former occurs just once). So it would seem the paper spells them "Parson's" but your surname is "Parsons"!

Dale: *There is quite a story behind this. I would like to retract that title right now and take the apostrophe off! I consider them my gift to the world, I prefer no ownership of them. What actually happened, is that was not the name of the paper. They were originally just called Programming Puzzles. We wrote the paper and at the last minute, Patricia thought it wasn't a jazzy enough title. She likes a bit of alliteration, so she thought Parsons Programming Puzzles would be better. We laughed about it, and the deadline was nearly up, so we did a search and replace quickly. We didn't proof read it at that point. The apostrophe was a mistake, there should be no apostrophe.*

Q2 Do you recall if there was any discussion about naming the puzzles? How close were they to being called Haden's Programming Puzzles?

Dale: *Patricia was our research coordinator. I work at a Polytech [type of university] and we had to be research active. Patricia was thinking what could we publish a paper on. I was thinking nothing, but I got these little programming puzzles. Patricia said write about them. They were my puzzles and it was my idea, but Patricia is actively brilliant, and she turned it into the paper. From that point on we had the most fantastic partnership. I would come up with ideas and experiment in the classroom, and Patricia would maneuver it into a good study and write the final paper. She loved writing and I hated it. We always thought the other one was doing all the work. It was a perfect partnership. I really want to emphasize how important Patricia's role was, there would not have been a paper without her. She is one of the finest academic writers I have ever come across. We were in a research partnership for 15 years and she was always the most valuable member of the team.*

Q3 Your paper motivates the idea well, but do you remember how you came up with it? What was the inspiration? Did you explore this activity in other ways before actually building the tool (based on Hot Potatoes)?

Dale: *My thing was that I would run the labs and during the lab the students would never make errors that I hadn't seen before. I was big on, oh, here we go again, it is the same error. I kept wanting to highlight them. We do a lot of showing code on the board, and asking what is wrong with this line, it is a very common error. That inspired me to do the puzzles. I was trying to use all that intel I had with all the common errors, that is what I was trying to do with the puzzles. I was hoping students would see that common error, and realize that is not right, I should be doing this, hopefully when they see the two lines together. It has always been disappointing that people didn't see it that way. People thought you were trying to trick them or distract them with the puzzles, but actually I was trying to point out common errors. I think that has been the least successful part of the puzzles. For implementing the puzzles, hot potatoes was my first attempt at implementing it. Later we looked at GUIs and other things and they were much prettier. Hot potatoes was just so easy, and made them so fast to create.*

Q4 How successful were Parsons problems in your own teaching? Did you find that they worked well and helped develop the skills you wanted students to learn? What lessons would you share with someone adopting Parsons problems for the first time?

Dale: *I think they were always for my struggling students, and they tended to be the ones that wanted to do them. They didn't really take off with my more advanced students, and the struggling students always wanted more of them. It was a constant struggle to make enough, and I never really had enough of them. But I always wondered, I always used them after the lab, as homeworky-type things, and I also wondered if they would have been better before the lab, so those students would have had an idea of what was to come, but I never tried that. I always put a lot of effort into those distractor lines, that was the whole purpose of the puzzle to me. I didn't think at the time that the code ordering was as important as the "let's don't fall for this common error." I was obsessed with the errors all the time. Now I think the code ordering has become more important and maybe more useful for the students. For advice for those adopting for the first time, I would still put a lot of effort into the errors, the distracting line, and loads of similar problems, because I still think you are aiming for those struggling students. I don't think the really good students needed them other than it is quick. But I think reading good lines of code and ending up with an exemplar, because the poor students didn't end up ever having readable bits of code because they were so lost. So I think it was beneficial to everyone. You trick them into reading code in the end.*

Q5 You told us that you already used Parson's problems post-lab and thought about using them pre-lab. Where and when would you like to see Parsons problems introduced into a student's learning trajectory?

Dale: *Well, I still think early on, very early in the course. Even if you had a bad day in the lab, then at least you could achieve these things, and you end up with a little bit of code that looks like it is going to work. It is all about making you feel better about your code. Nothing worse than spending an hour in the lab and nothing works. And the students found them fun, so it was all about motivating them to keep going.*

Q6 The use of Activity Diagrams (a bit like flowcharts) were an important theme in the approach you reported in your paper. These have been less commonly used by others, in favour of textual problem prompts. Did you continue to use such diagrams over time?

Dale: *I love them. We still teach activity diagrams in the first computing course. I think it is probably an old fashioned thing. When I was learning to program, we would have a whole course where you actually wouldn't code, you would just do an activity diagram or flow chart. And you weren't allowed to code until you had completed your flow chart. And I always liked it as a visual representation of the code. It didn't matter what language you were going to use, the activity diagram would be the same. So we thought it was a natural thing that you could give an activity diagram and make them into a Parsons problem from it. And we still use them for exam questions. I guess that didn't catch on, but I still like that.*

Q7 Do you have any particular memories (good or bad) of using Parsons problems in your own classroom or with individual students?

Dale: *I thought the students liked them. That was a big thing for me. They liked doing them and they would ask for more. And I thought, well they very seldom ask for more code problems, so I thought that was a good thing. In the early days, I distinguished between the puzzles and the problems, so to me, a Parsons problem is an exam question and the puzzles is always the electronic rearranging. I just thought, in the early days when I first came across the problems, I thought, oh what a shame. Here was my learning tool for the struggling students that has now become this thing that might fail them. In the end, we probably used the exam questions more than the puzzles.*

Q8 Was it immediately clear that you would write a paper about Parsons problems? It is common in our field for a lot of good practice to remain unpublished.

Dale: *A lot of really good people are doing practice in the classroom and there are not many avenues for publishing it if it is not a scientific study. We would have all these ideas, and Patricia would go, "Oh, it is just another tool". But we need a place to show those tools, because that was all that this was. It was a little drag and drop puzzle, that I didn't think would go anywhere, until Patricia put a bit more of an academic spin on it, and then we got published in ACE. I think there must be heaps of little ideas and little tools out there that we need to spread, but when you have got to have it as a big study and when you are a* *little Polytech, it is really hard to get stuff published where people were going to see it. So I was surprised that people picked it up or found it. Everybody needs a Patricia Haden! They need someone to go OOh, OOh, other people will be interested in this. And we need conferences where people can show that stuff. When we went to Hobart, people liked the paper, but nobody thought it was going to go anywhere. It was a successful little paper presentation, but I don't think any of us thought, this has got legs.*

Q9 Your original paper was published at ACE (which is a local conference). How important to you was it to have this local outlet for your work?

Dale: *This paper was published at ACE which was at Hobart and that was part of the attraction, we wanted to go to Tasmania. At the conference, I remember people were very kind. It was a very supportive conference. I didn't know anyone but Patricia, and they were very welcoming. Patricia had been in previous years. That sort of group was very manageable and very inclusive, we couldn't have done it without having a conference like that. It is still going isn't it. We really needed it. We wouldn't have been able to get a little idea like that to a bigger conference at the time. It was just a hot potatoes drag and drop puzzle. That was always our go to conference. When Patricia came around and would say, you got to do something, that is where we would aim for. And we loved the tourism around Australia.*

Q10 How has it been for you to have your name associated with these problems, which have become very popular in computing education?

Dale: *Professionally it has been really good, it has helped me travel, and people take me seriously in our own institution. So that has all been good. But, it was an accident, and we still just call them programming puzzles, until exam questions came along and that is what we call Parsons problems. I wouldn't recommend it, and it is all Patricia's fault.*

Q11 Did you expect that Parsons problems would become so popular? What were your expectations when you wrote the original paper?

Dale: *No, No. I was embarrassed about the hot potatoes, because I thought, it was just so easy. So we threw that out and we designed that little GUI interface. Just because we were embarrassed about how easy the hot potatoes and how amateurish it looked. Later we did use the JSparsons, and there was a guy in Canada that was making a teaching one, but they were just too slow to make. What I really needed and never really got, was I wanted to copy my code into something and then tinker with it later. It took me too long to make them.*

Q12 What are your thoughts on the many novel tools and adaptations that have been applied to Parsons problems (e.g. faded parsons problems, design-level parsons problems, adaptive parsons problems, etc.)

Dale: *Yeah, I love it. I love those, the faded Parsons problems, where you wrote into them as well. I thought, Ooh Ooh, that is good. It has gone a long way from the beginner tool, hasn't it. Now it is looking like a tool for everyone. I love Barbara's work with her adaptive problems. And I haven't come across the design-level ones, so I must look those up. It is weird when you are not in Academia any more, you don't have access to papers, so I occasionally stumble over something on YouTube and watch them that way. So I put out a plea, that they make them all publicly available so I can read them.*

Q13 Is there anything you originally conceived of with Parsons problems that hasn't been tried yet, or that you would like to see done? Are there any particular lines of inquiry you would like to see researchers explore?

Dale: *I wonder about grading them. I still think they are ideally suited to the struggling student. And it is like there is a spectrum now. I think the faded problems are probably on the hard side of the spectrum. And I still can't help, maybe I should let it go, but the common errors, that never took off. Nobody could ever prove that they were useful. I still thought always that was my motivating factor. All those mistakes I had seen over all those years, I just thought that would be useful. I don't think it was proved that that was useful. Now for distractors, just the word distractor sounds like we are trying to trick them. Actually, the whole idea was, "Ooh, Ooh, I nearly fell for that. I'm pretty sure that is not right." That didn't take off and people haven't really used that how I thought they would. I was just trying to get all that intel I had and put it into the puzzle. My final message is tell them there is no apostrophe. We blame search and replace.*

C APPENDIX: SLR INCLUDED ARTICLES

Table 11: Systematic Literature Review: Final Set of Included Articles

Year	Title	Reference
2006	Parson's programming puzzles: A fun and effective learning tool for first programming courses	[177]
2008	Evaluating a New Exam Question: Parsons Problems	[43]
2008	Relationships between reading, tracing and writing skills in introductory programming	[142]
2009	Surely we must learn to read before we learn to write!	[141]
2010	A proposal for a new communication medium in the classroom	[135]
2010	An introduction to program comprehension for computer science educators	[195]
2010	Instructor perspectives of multiple-choice questions in summative assessment for novice programmers	[200]
2010	Learning (through) recursion: A multidimensional analysis of the competences achieved by CS1 students	[160]
2010	Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer	[138]
2010	Wearing the assessment 'BRACElet'	[212]
2011	Two-dimensional parson's puzzles: The concept, tools, and first observations	[109]
2012	A mobile learning application for parsons problems with automatic feedback	[119]
2012	How do students solve parsons programming problems?: an analysis of interaction traces	[101]
2013	CS circles: an in-browser python course for beginners	[188]
2013	How Do Students Solve Parsons Programming Problems? – Execution-Based vs. Line-Based Feedback	[100]
2013	How to study programming on mobile touch devices: interactive Python code exercises	[107]
2014	Research-based design of the first weeks of CS1	[205]
2015	Analysis of Interactive Features Designed to Enhance Learning in an Ebook	[60]
2015	Comparing student performance between traditional and technologically enhanced programming course	[117]
2015	Designing for deeper learning in a blended computer science course for middle school students	[91]
2015	Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles	[96]
2015	Interactive Learning Content for Introductory Computer Science Course Using the ViLLE Exercise Framework	[118]
2015	Learning in Distributed Low-Stakes Teams	[146]
2015	PILeT: an Interactive Learning Tool To Teach Python	[7]
2015	Structuring Flipped Classes with Lightweight Teams and Gamification	[136]
2015	Tasks that can improve novices' program comprehension	[199]
2015	Usability and Usage of Interactive Features in an Online Ebook for CS Teachers	[58]
2015	What are we doing when we assess programming?	[178]
2016	Applying Validated Pedagogy to MOOCs: An Introductory Programming Course with Media Computation	[74]
2016	Automatically assessed electronic exams in programming courses	[190]
2016	Combining parson's problems with program visualization in CS1 context	[202]
2016	Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers	[95]
2016	Identifying Design Principles for CS Teacher Ebooks through Design-Based Research	[64]
2016	Investigating strategies used by novice and expert users to solve parsons problems in a mobile python tutor	[67]
2016	Learning Loops: A Replication Study Illuminates Impact of HS Courses	[163]
2016	Learning programming from tutorials and code puzzles: Children's perceptions of value	[94]
2016	M-Learning: A New Paradigm of Learning ICT in Nigeria	[170]
2016	Ne-course for learning programming	[78]
2016	Reading Hierarchies in Code: Assessment of a Basic Computational Skill	[175]
2016	Redesigning an Object-Oriented Programming Course	[116]
2016	Subgoals Help Students Solve Parsons Problems	[165]
2017	A comparison of different types of learning activities in a mobile python tutor	[69]
2017	An Exploratory Study of the Usage of Different Educational Resources in an Independent Context	[102]
2017	Evaluation of PILeT: Design guidelines, usability and learning outcomes results	[6]
2017	Evolutionary practice problems generation: More design guidelines	[85]
2017	Improving online learning activity interoperability with acos server	[203]

2017	Integrating Parson's Programming Puzzles into a game-based mobile learning application	[169]
2017	Investigating the effectiveness of menu-based self-explanation prompts in a mobile Python tutor	[70]
2017	Learning with engaging activities via a mobile Python tutor	[71]
2017	Precursor skills to writing code	[233]
2017	Refactoring a CS0 course for engineering students to use active learning	[139]
2017	Solving Parsons Problems Versus Fixing and Writing Code	[62]
2017	Students and Teachers Use An Online AP CS Principles EBook Differently: Teacher Behavior Consistent with Expert Learners	[176]
2017	Teaching robotics concepts to elementary school children	[81]
2017	The Code Mangler: Evaluating Coding Ability Without Writing any Code	[34]
2017	The Effect of Providing Motivational Support in Parsons Puzzle Tutors	[127]
2017	Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw	[38]
2017	What do novices think about when they program?	[5]
2018	Active Learning through Game Play in a Data Structures Course	[47]
2018	Adaptive Problem Selection in a Mobile Python Tutor	[72]
2018	An integrated practice system for learning programming in Python: design and evaluation	[31]
2018	Epplets: A Tool for Solving Parsons Puzzles	[128]
2018	Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems	[59]
2018	PCEX: Interactive Program Construction Examples for Learning Programming	[105]
2018	Student performance prediction by discovering inter-activity relations	[193]
2018	Supporting novices and advanced students in acquiring multiple coding skills	[68]
2018	Ten quick tips for teaching programming	[29]
2018	Tracing vs. Writing Code: Beyond the Learning Hierarchy	[97]
2019	A Spaced, Interleaved Retrieval Practice Tool that is Motivating and Effective	[231]
2019	An analysis of interactive feature use in two ebooks	[54]
2019	Board 59: Coevolutionary-Aided Teaching: Leveraging the Links Between Coevolutionary and Educational Dynamics	[86]
2019	Crossing the borders: Re-use of smart learning objects in advanced content access systems	[152]
2019	E-Assessment in Programming Courses: Towards a Digital Ecosystem Supporting Diverse Needs?	[35]
2019	Effective and Innovative Interactives for icseBooks	[41]
2019	Evaluating the Effectiveness of Parsons Problems for Block-based Programming	[234]
2019	Evaluation of Parsons Problems with Menu-Based Self-Explanation Prompts in a Mobile Python Tutor	[73]
2019	EvoParsons: design, implementation and preliminary evaluation of evolutionary Parsons puzzle	[11]
2019	Experience Report: Thinkathon – Countering an "I Got It Working" Mentality with Pencil-and-Paper Exercises	[40]
2019	Helping Students Solve Parsons Puzzles Better	[129]
2019	Impact of Puzzle-based Learning Technique for Programming Education in Nigeria Context	[168]
2019	Investigating the affect and effect of adaptive parsons problems	[57]
2019	Lessons learned from available parsons puzzles software	[87]
2019	Mnemonic Variable Names in Parsons Puzzles	[130]
2019	On the potential of evolved Parsons puzzles to contribute to concept inventories in computer programming	[12]
2019	Position: Scaffolded Coding Activities Afforded by Block-Based Environments	[145]
2019	Predicting Cognitive Load in Future Code Puzzles	[123]
2019	Representing and Evaluating Strategies for Solving Parsons Puzzles	[131]
2019	Social Worked-Examples Technique to Enhance Student Engagement in Program Visualization	[3]
2019	Speed and Studying: Gendered Pathways to Success	[230]
2019	Teaching computer programming with PRIMM: a sociocultural perspective	[196]
2020	A Review of Research on Parsons Problems	[48]
2020	Advanced Comprehension Analysis Using Code Puzzle: Considering the Programming Thinking Ability	[112]
2020	Analyzing Parsons Puzzle Solutions using Modified Levenshtein's Algorithm	[148]
2020	Bugs as Features: Describing Patterns in Student Code through a Classification of Bugs	[151]
2020	Code Writing vs Code Completion Puzzles: Analyzing Questions in an E-exam	[201]
2020	Comprehension analysis considering programming thinking ability using code puzzle	[113]
2020	Crescendo: Engaging Students to Self-Paced Programming Practices	[222]
2020	CSAwesome: AP CSA curriculum and professional development (practical report)	[56]

2020	Exploring Student-Controlled Social Comparison	[1]
2020	Generic Tasks for Algorithms	[157]
2020	Improving Engagement in Program Construction Examples for Learning Python Programming	[104]
2020	Infusing Computing: A Scaffolding and Teacher Accessibility Analysis of Computing Lessons Designed by Novices	[33]
2020	Introducing a Paper-Based Programming Language for Computing Education in Classrooms	[156]
2020	Runestone: A Platform for Free, On-line, and Interactive Ebooks	[63]
2020	Split-paper testing: A novel approach to evaluate programming performance	[167]
2020	The curious case of loops	[164]
2020	Using Edit Distance Trails to Analyze Path Solutions of Parsons Puzzles	[149]
2020	Work in Progress: Parsons Problems as a Tool in the First-Year Engineering Classroom	[162]
2021	A Simple, Language-Independent Approach to Identifying Potentially At-Risk Introductory Programming Students	[17]
2021	A Tool Help for Introductory Programming Courses	[76]
2021	An online platform for teaching upper secondary school computer science	[220]
2021	Analysis of the Answering Processes in Split-Paper Testing to Promote Instruction	[217]
2021	Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose	[39]
2021	Beyond Programming: A Computer-Based Assessment of Computational Thinking Competency	[134]
2021	Code Beats: A Virtual Camp for Middle Schoolers Coding Hip Hop	[143]
2021	Collaborative Parsons Problems in a Remote-learning First-year Engineering Classroom	[161]
2021	Comparing Ebook Student Interactions With Test Scores: A Case Study Using CSAwesome	[147]
2021	Do Students Use Semantics When Solving Parsons Puzzles? A Log-Based Investigation	[132]
2021	Enhancing postgraduate students' technical skills: perceptions of modified team-based learning in a six-week multi-subject Bootcamp-style CS course	[22]
2021	Estimating Learner's Perspective in Programming:Analysis of Operation Time Series in Code Puzzles	[114]
2021	Evaluating Parsons Problems as a Design-Based Intervention	[82]
2021	Improving Instruction of Programming Patterns with Faded Parsons Problems	[225]
2021	Integrating Diverse Learning Tools using the PrairieLearn Platform	[226]
2021	Integrating Parsons Puzzles with Scratch	[20]
2021	Investigating the Impact of Computing vs Pedagogy Experience in Novices Creation of Computing-Infused Curricula	[111]
2021	Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code	[98]
2021	SQL Scrolls - A Reusable and Extensible DGBL Experiment	[189]
2021	Student Practice Sessions Modeled as ICAP Activity Silos.	[88]
2021	Teaching and Learning Strategies for Introductory Programming in University Courses	[79]
2021	Teaching and learning tools for introductory programming in university courses	[77]
2021	Teaching CS Middle School Camps in a Virtual World	[174]
2021	Tutors' Experiences in Using Explicit Strategies in a Problem-Based Learning Introductory Programming Course	[90]
2021	Using a Computer to Score Parsons Problems Answered on Paper	[206]
2021	Using Markov Transition Matrix to Analyze Parsons Puzzle Solutions	[133]
2021	When Wrong is Right: The Instructional Power of Multiple Conceptions	[155]
2022	Detecting Struggling Students from Interactive Ebook Data: A Case Study Using CSAwesome	[61]
2022	Enhancing problem-solving skills of novice programmers in an introductory programming course	[150]
2022	Investigating Students' Performance and Motivation in Computer Programming through a Gamified Recommender System	[2]
2022	Process-Oriented Understanding Estimation Using Code Puzzles	[115]
2022	Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study	[80]
2022	Retrieval-based Teaching Incentivizes Spacing and Improves Grades in Computer Science Education	[229]