Improving Instruction of Programming Patterns with Faded Parsons Problems

Nathaniel Weinman
nweinman@berkeley.edu
UC Berkeley

Armando Fox
fox@berkeley.edu
UC Berkeley

Marti A. Hearst
hearst@berkeley.edu
UC Berkeley

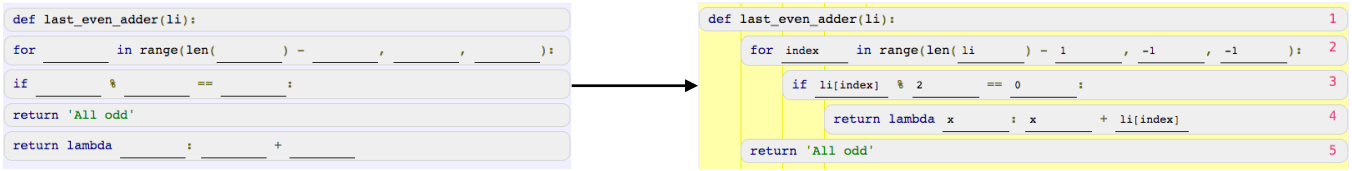

Figure 1: An example of an unsolved Faded Parsons Problems (left) and the completed solution (right) using a Premature Return pattern to solve a Higher-Order Function problem. Students solve the question by both rearranging and completing the given lines of code.

ABSTRACT

Learning to recognize and apply programming patterns — reusable abstractions of code — is critical to becoming a proficient computer scientist. However, many introductory Computer Science courses do not teach patterns, in part because teaching these concepts requires significant curriculum changes. As an alternative, we explore how a novel user interface for practicing coding — Faded Parsons Problems — can support introductory Computer Science students in learning to apply programming patterns. We ran a classroom-based study with 237 students which found that Faded Parsons Problems, or rearranging and completing partially blank lines of code into a valid program, are an effective exercise interface for teaching programming patterns, significantly surpassing the performance of the more standard approaches of code writing and code tracing exercises. Faded Parsons Problems also improve overall code writing ability at a comparable level to code writing exercises, but are preferred by students.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → Patterns.

KEYWORDS

Computing Education, CS1, Programming Patterns, Parsons Problems

ACM Reference Format:
Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3411764.3445228

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445228

1 INTRODUCTION

Despite the increasing relevance of programming [2, 5, 31], developing this skill continues to be difficult for novices to learn in part because programming is a complex task consisting of many skills and types of knowledge [3, 9, 17].

One such skill, the ability to recognize and use programming patterns, is a distinguishing characteristic between experts and novices [29]. *Programming patterns* are partial implementations of reusable, higher-level concepts which can achieve a goal. For example, the solution in Figure 1 represents a "Loop: Premature Return" pattern, searching the list backwards and returning a higher-order function as soon as an even number is found. This pattern consists of a conditional return within a loop, with a catch-all return outside of that loop. This same pattern, with different code, could be used for many similar purposes, such as to find the first occurrence of a character in a string or the largest even number in an increasingly sorted list. These patterns provide the building blocks that allow programmers to efficiently tackle more complex tasks. However, despite their importance, they are often not explicitly taught in CS1 classrooms [25].

Programming patterns can be challenging to teach because they rely on a foundation of conceptual knowledge [37]. For example, the pattern in Figure 1 requires understanding of `for` loops and `range()`. Researchers have developed methods to teach programming patterns, but these involve significant modifications to the structure of the course curriculum [19, 20, 22, 30, 37]. For example, Pattern-Oriented Instruction [25], as part of their guidelines, proposes discussions after all problem-solving activities and prepared material to compare patterns after complex exercises. The practical difficulties of introducing large-scale innovations to classroom teaching suggest that an approach that minimizes changes to the status quo will lead to better adoption over major structural changes.

Most computer programming classes already make use of assigned programming exercises, highlighting opportunity for easily integrated improvements. Many programming courses use a combination of code tracing exercises, in which students predict the behavior of instructor-provided code, and code writing exercises, in which students write code in response to an exercise prompt. In this work, we compare these commonly used exercise interfaces, as well as a recently-introduced exercise interface, Faded Parsons Problems (Figure 1) [34], as a low-friction way to teach programming patterns. In this paper, we emphasize exercise interfaces as one way to distinguish between different types of exercise. We find that the content of existing code tracing and code writing exercises can be easily re-purposed as Faded Parsons Problems.

Faded Parsons Problems are a particular variation of Parson Problems [26]. In Faded Parsons Problems, creators give students a set of partially complete lines of code. Students must fill in blanks in the lines of code with valid expressions as well as rearrange the lines of code to construct a valid program. Unlike code tracing, with Faded Parsons Problems, students must actively construct the structure and logic of a program. However, students are heavily constrained by the given lines of code, thus deliberately excluding many valid alternative solutions to the exercise.

We ran an in-situ study as part of an introductory Computer Science class and found strong evidence to confirm that Faded Parsons Problems are effective at teaching programming without any modifications to instruction. Our study also explores how Faded Parsons Problems affect transfer to student code writing ability in general, as well as qualitative data to understand how students engage with Parsons Problems and limitations of code writing exercises.

Our contributions are as follows:

- We deploy a system which supports a range of programming exercise interfaces which scales to a class of hundreds of students and is well-suited for remote learning.
- We confirm the efficacy of Faded Parsons Problems in teaching students to recognize and apply programming patterns to relevant exercise prompts.
- We confirm that practicing with Faded Parsons Problems produces similar transfer to code writing ability as directly practicing with code writing exercises.
- We probe student attitudes towards Faded Parsons Problems, finding students prefer working with Faded Parsons Problems when given a choice and providing insights as to why.

2 RELATED WORK

2.1 Programming Patterns

Patterns (or plans, schemas, templates) have several subtly different definitions in the literature [1, 8, 21, 25, 33, 36, 37]. In general, they are higher-level, reusable abstractions of code that achieve a specific goal. Patterns are also hierarchical and multi-layered, with some smaller patterns contained in other larger ones. The importance of patterns is supported through chunking from cognitive theory, in which people, as they view examples with identifiable similarities, construct and store more complex patterns as single cognitive "chunks" [6, 21].

Studies have found evidence that one way the behavior of expert programmers is different from novices is in their ability to leverage patterns in understanding and writing code [29, 36]. However, patterns remain out of the focus in many Computer Science classes, leading to proposals for significant instructional shifts to address this concern.

Muller et al. [25] propose Pattern-Oriented Instruction in introductory Computer Science classes, where patterns are explicitly incorporated into a course's instruction. They found that this explicit change to instruction led to novices improving their problem decomposition and solution construction skills. Xie et al. [37] also designed a curriculum focused on explicit instruction of templates. They find some evidence that this new template-oriented curriculum improved student coding ability.

Xie et al. [37] also note the importance of learning syntactic or conceptual knowledge before learning patterns. That is, students must understand all the building blocks of a pattern before being able to understand the pattern itself, and similarly for writing building blocks and patterns. That is, though patterns are a foundational skill for programming, they are not entirely introductory. This may explain why many CS1 classes today still do not focus on explicitly teaching patterns.

We build on the existing work showing that patterns are both important and could be taught better in most CS1 classrooms. Our work differs, however, in its focus on a much simpler instructional change. Our work focuses on assigned programming exercises – a part of the course where students already spend considerable time. Instead of updating curricula to explicitly teach patterns to students, we focus on whether students can recognize and incorporate patterns more effectively by working on programming exercises with different user interface.

2.2 User Interfaces for Program Exercises

Several frameworks have been proposed for conceptualizing user interfaces for programming exercises. Cutts et al. [9] propose that programmers must master moving between three levels of abstraction: English, pseudocode, and code, proposing different exercise interfaces to target different levels of abstraction. Bryant et al. [4] extend this further, proposing five levels of abstraction. They posit that expressing algorithms in human language, when compared to code, changes how programmers engage with their programming task, suggesting that pair programming causes programmers to focus more on an intermediate level of abstraction compared to when programming alone.

Shi et al. [32] motivate their work from a different perspective, focused on the different stages of problem solving. They design Pyrus, an interface where students work together with limited knowledge to focus student attention and growth on the planning stage of the problem-solving process. By constraining students' actions in solving the exercise, they find students spend more time planning. Python Tutor [15] has become a popular educational tool for introductory students because its granular visualizations and debugging functionality are well-suited to novice programmers.

Our work is motivated by these frameworks and interfaces, exploring how the design behind programming exercise interfaces affects how students engage with the content. Thoughtfully considering

these designs can support programmers in more effectively practicing specific programming knowledge or skills. Our work reveals insight into our recently introduced interface, Faded Parsons Problems [34].

2.3 Parsons Problems

Parsons Problems (or Parsons Puzzles, code scrambles, Code Mangler Problems) require students to unscramble lines of code to construct a syntactically and logically correct program. They were originally designed to be an engaging task for students to practice syntax drills [26]. They share many similarities with block-based programming languages such as Scratch, Snap!, Alice, and Blockly, which provide an engaging introduction to certain introductory topics [23, 28, 35]. However, unlike block-based languages, Parsons Problems support the full expressivity of text-based programming languages such as Python.

Zavala and Mendoza [38] found support that code comprehension (e.g., code tracing exercises), code manipulation (e.g., Parsons Problems), and code writing are separate skills that should be mastered in that order. Ericson et al. [12] successfully integrated Parsons Problems into teaching, finding that Parsons Problems, when combined with worked examples, provide similar learning gains to code writing or bug detection in a CS1 classroom while taking only 70% of the time. Ericson et al. [11] extended this study, finding similar results for Adaptive Parsons Problems, which dynamically adjust difficulty by adding or removing unnecessary or incorrect lines of code.

However, Denny et al. [10] raise concerns that Parsons Problems may be easily "gamed" by sufficiently mature students using syntactic heuristics. In our formative study for the present work, a student described their strategy for solving Parsons Problems as "just kind of moving things around based on test cases, not really thinking about the logic." In previous work [34], we found that solving a Parsons Problem did not transfer to being able to write code for the same question, and attempted to address this limitation by introducing Faded Parsons Problems. In Faded Parsons Problems, provided lines of code can be partially or fully incomplete. Faded Parsons Problems integrate code writing with Parsons Problems, providing less syntactic and logic scaffolding than standard Parsons Problems.

The present work differs from previous work on Parsons Problems two ways. First, this work primarily focuses on Faded Parsons Problems. Second, this work focuses on teaching programming patterns, a targeted subset of general programming ability.

We focus on three interfaces. Code tracing exercises (Figure 2), in which students predict the behavior of instructor-provided code, require students to *understand* a *well-designed solution*. Code writing exercises (Figure 3), in which students write their own code in response to an exercise prompt, require students to *construct* a solution, though possibly a poor one. Faded Parsons Problems require students to *reconstruct* a *well-designed solution* by using constraints to both lock out valid solutions and scaffold the solving process.

3 PROGRAMMING PATTERNS

Following existing definitions in the literature, we define **programming patterns** as partial implementations of reusable, higher-level concepts which can achieve a goal. The programming patterns used in this study are specific to Python and are based in coding structures.

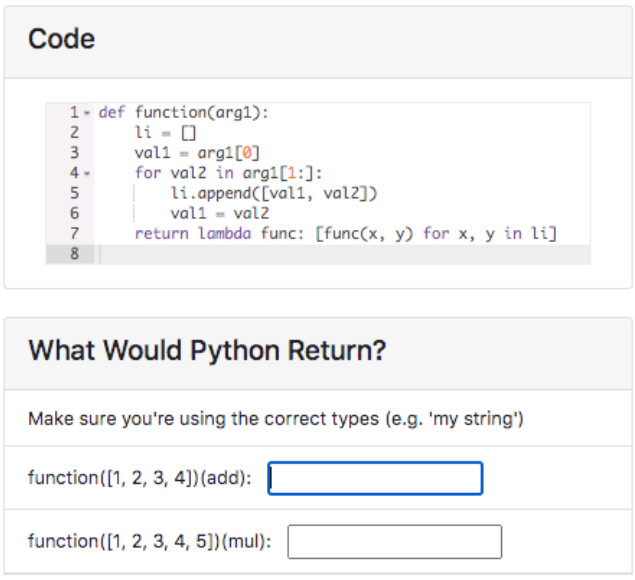

Figure 2: A code tracing exercise. Students must read the obfuscated code (top) and determine the output from specified argument prompts (bottom)

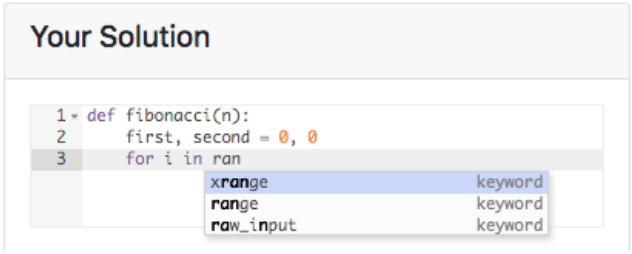

Figure 3: A code writing exercise. Students must create a functioning program to solve a given exercise prompt using an editor with basic IDE functionality like syntax highlighting and auto-completion.

We ground our definition of programming patterns in actual code to ensure they contain certain language idioms, some of which are specific to Python. For example, L2 (defined in Table 1) includes the use of `enumerate(li)` in a `for` loop. A more abstract, language-agnostic definition would include more verbose implementations such as iterating through `range(len(li))` and then also defining a local variable from indexing into the list. This example also highlights how patterns must be built on both conceptual knowledge, e.g., understanding loops, and syntactic knowledge, e.g., `enumerate`. Some of the syntactic features on which patterns are built, such as `for`, may be more fundamental than others.

The pattern adherence of a given program can be evaluated both by the use of control flow elements as well as specific syntactic

elements. An advantage of this is that patterns are specific enough to be effectively captured with automated parsing logic, allowing efficient analysis across tens of thousands of submissions. Additionally, we can check the pattern adherence of a program even if the program contains unrelated syntax errors, which happens frequently for introductory student submissions.

Table 1 lists the programming patterns used in this study, along with definitions. These programming patterns were selected by analyzing programming assignments from previous versions of two introductory Computer Science courses. That is, existing course materials, developed by the instructor, determined which programming patterns were used in this study.

3.1 Examples of Programming Patterns

The patterns were selected so the three Loop patterns, L1-L3, would appear appropriately early in the course, focusing on foundational material. Though these patterns were used in exercises over multiple weeks, tangential concepts covered within the exercises changed over the course of the semester. For example, lambdas and higher order functions were introduced into the patterns later in the semester. The example in Figure 1, for instance, returns a function which takes an argument `x` and adds it to the last even element in the given list. Another exercise using this pattern earlier in the course returned True if any numbers in a list are divisible by five. The next two patterns (MR1, MR2) were selected to match course content on multiple recursion, focusing on algorithmically challenging exercises. The final two patterns (OOP1, OOP2) included class definitions in order to match course content on Object-Oriented Programming, focusing on syntactically and conceptually challenging exercises. These patterns were chosen to cover a range of different aspects of programming.

Table 1: Examples of programming patterns and their descriptions.

Goal/Name	Description
Loop: Premature Return (L1)	A return within a loop based on a conditional, with a final return outside of that loop
Loop: Index Value (L2)	Use of `enumerate` to access and update both the index and value of elements, using both inside the loop.
Loop: Last/Current (L3)	Inside a loop, use last and then update last to current.
Stateful Tree Traversal (MR1)	Traverse a Tree in depth-first-search using a default argument to pass state.
Multiple Recursion Game Simulation (MR2)	Check for a base case, simulate moves, then return based on the results.
Stateful Generators (OOP1)	Define generators as part of a class for stateful instance variables, yielding at the end of loops.
Mixins (OOP2)	Use `super` to access parent methods.

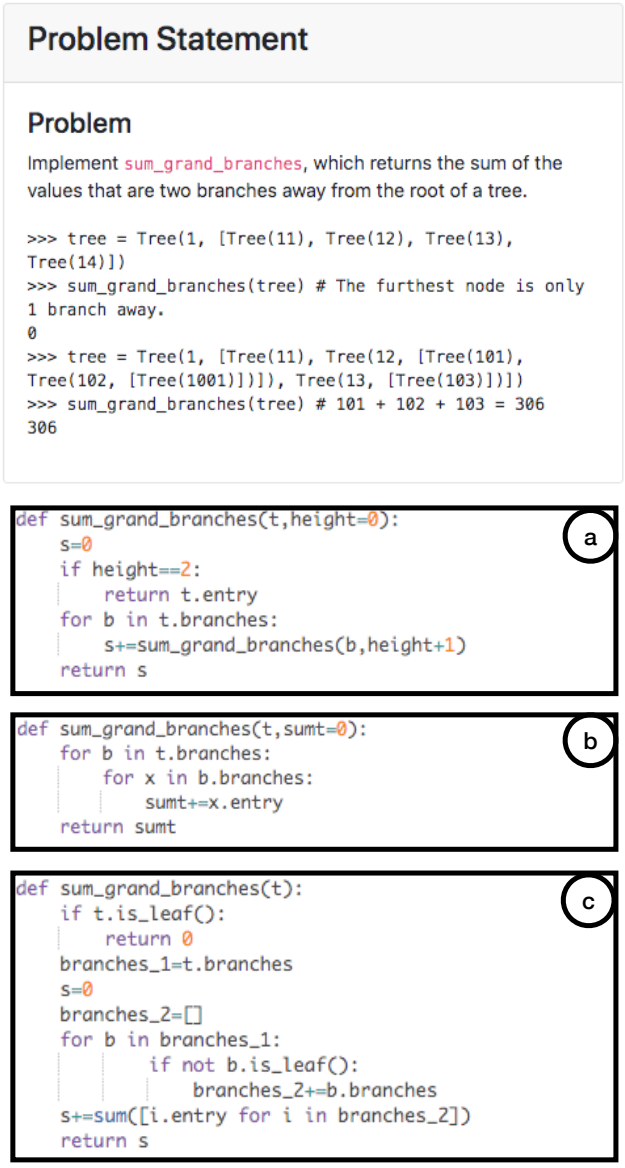

Figure 4: Three different correct student submissions to an exercise for which students could apply the MR1 pattern. (a) A reasonable solution to the exercise which follows the pattern. (b) A reasonable solution to the exercise which does not follow the pattern. (c) An overly complex solution to the exercise which does not follow the pattern.

3.2 Programming Patterns in Code

To better illustrate the role of programming patterns in student work, Figure 4 contrasts three correct student submissions to a problem — summing nodes of a Tree at depth 2 — in an open-ended code

writing exercise. These submissions were selected from the study described in Section 5. All students had immediately previously worked on a question computing the depth of a Tree of arbitrary height. Submission (a) follows the MR1 pattern. It is very similar to the instructor solution for the previous exercise and a reasonable solution to this one. Submission (b) is a reasonable submission that does not follow the pattern. Instead, it takes advantage of the fact that this exercise is asking about a fixed depth within the Tree, using nested for loops instead of recursion. While this is a valid solution, it defeats the instructor's motivation to have students recognize this exercise as an opportunity to practice recursion to navigate Trees. This highlights a challenge of teaching programming patterns, as there can sometimes be other reasonable solutions to specific exercises. Submission (c) is a correct submission which does not follow the pattern, and is overly complex and verbose. It is not uncommon for students to create poorly structured solutions such as these in auto-graded code writing exercises, due to the lack of granular feedback and the iterative manner with which students can create solutions.

Though code writing exercises are an effective form of practice for programming, the example above illustrates why they may be ill-suited for helping students learn how to use patterns as students only sometimes construct solutions following a relevant pattern. If students are given the freedom to solve two related exercises in entirely different ways, they miss an opportunity to compare and contrast two applications of the same pattern, an important opportunity to generalizing the abstraction. Code tracing exercises allow instructors to control the structure of code students are interacting with, providing a good opportunity for teaching patterns. However, they do not give students any practice in constructing algorithms or syntax to create a program.

4 USER INTERFACE FOR PROGRAMMING EXERCISE COMPARISON

We built a Flask app that extends Karavirta et al.'s js-parsons library [16] to support all exercise interfaces used in this study, including Faded Parsons Problems. The system supports Python programming exercises—traditional code writing, Faded Parsons Problems, code tracing, code skeleton, and multiple-choice comprehension—as well as short answer survey questions. A nearly complete Faded Parsons Problem exercise can be seen in Figure 5(c). In Faded Parson Problem exercises, the user is initially given a set of blocks containing partially incomplete code on the left including optional print and comment statements (d) for debugging, with the initial function signature populated on the right. The lines on the left are initially alphabetized, as suggested by Cheng and Harrington [7]. To solve a Faded Parsons Problem, users drag fragments in a correct order and indentation on the right and complete the blanks (e).

All exercises display the problem statement (b) above the interface. Users can run pre-configured tests as often as they want, which displays detailed output from the test cases (h) including: function arguments, expected output, actual output, any standard output from print statements, and any raised exceptions. The type of feedback is consistent across all programming interfaces. Not pictured: At any point, users can return to a list of exercises for that week. Additionally, users can view a correct solution to the exercise after solving the question or expending enough effort on the exercise based on custom time- and submission-based logic.

The Flask app logs anonymized data from participants, and randomizes treatment selection. The autograder is a separate worker that uses RQ[1] to safely execute arbitrary Python code, allowing execution-based feedback. Instructors manually configure the blanks in Faded Parsons Problems.

5 EVALUATION

We run a study to explore the following research questions:

RQ1: How does practice with different exercise interfaces affect student acquisition of programming patterns? To be able to learn a pattern, students must first be exposed to examples of it. We measured **Pattern Exposure** to see if students encounter pattern-adherent code as they craft their own solution or by viewing the instructor solution. We also measured **Active Pattern Exposure**, a subset of this, to see if students craft a pattern-adherent solution. In later exercises, we measured **Pattern Acquisition** to see if students, after being exposed to examples of a pattern, are able to recognize the opportunity and apply a pattern in a code writing exercise. We note that students can apply a pattern without correctly solving an exercise, such as the code in Figure 5.

RQ2: What is the general efficacy of practice with different exercise interfaces? This work is motivated by the ease in which instructors can replace existing exercises with different exercise interfaces. To that end, we must understand the educational impact of different exercise interfaces beyond patterns. Since code writing exercises are often used as a de facto measurement of programming expertise, we measured **Code Writing Transfer** to see if students can successfully transfer [27] what they learned in each interface to similar code writing questions regardless of any pattern use.

RQ3: What is student perception of working with Faded Parsons Problems? In several exercises, students were able to select which exercise interface they worked with, which implicitly suggests preference. Students were also asked Likert-scale and open-ended survey questions to understand their **Preference** surrounding Faded Parsons Problems. These survey questions also provide insight into the **Perceived Difficulty** of these exercise interfaces, which we compare to the **Actual Difficulty**.

5.1 Study Environment and Participants

We partnered with an instructor of a CS1 class at a large US research university with IRB approval to run our evaluation study in three parts. Throughout the course of the semester, 237 students (P1-P237, 111 male, 120 female, 6 unreported) agreed to the Terms of Consent and interacted with our system. The study was deployed as extra credit assignments appended to 10 of the 12 weekly lab assignments. Students could begin the questions in an in-person lab, but had a week to continue working on the exercises on their own. The extra credit exercises were effort-based (as opposed to completion-based) to better match how labs were assessed. The instructor approved all exercises used in the study, but was not aware of the specific

[1] https://python-rq.org/

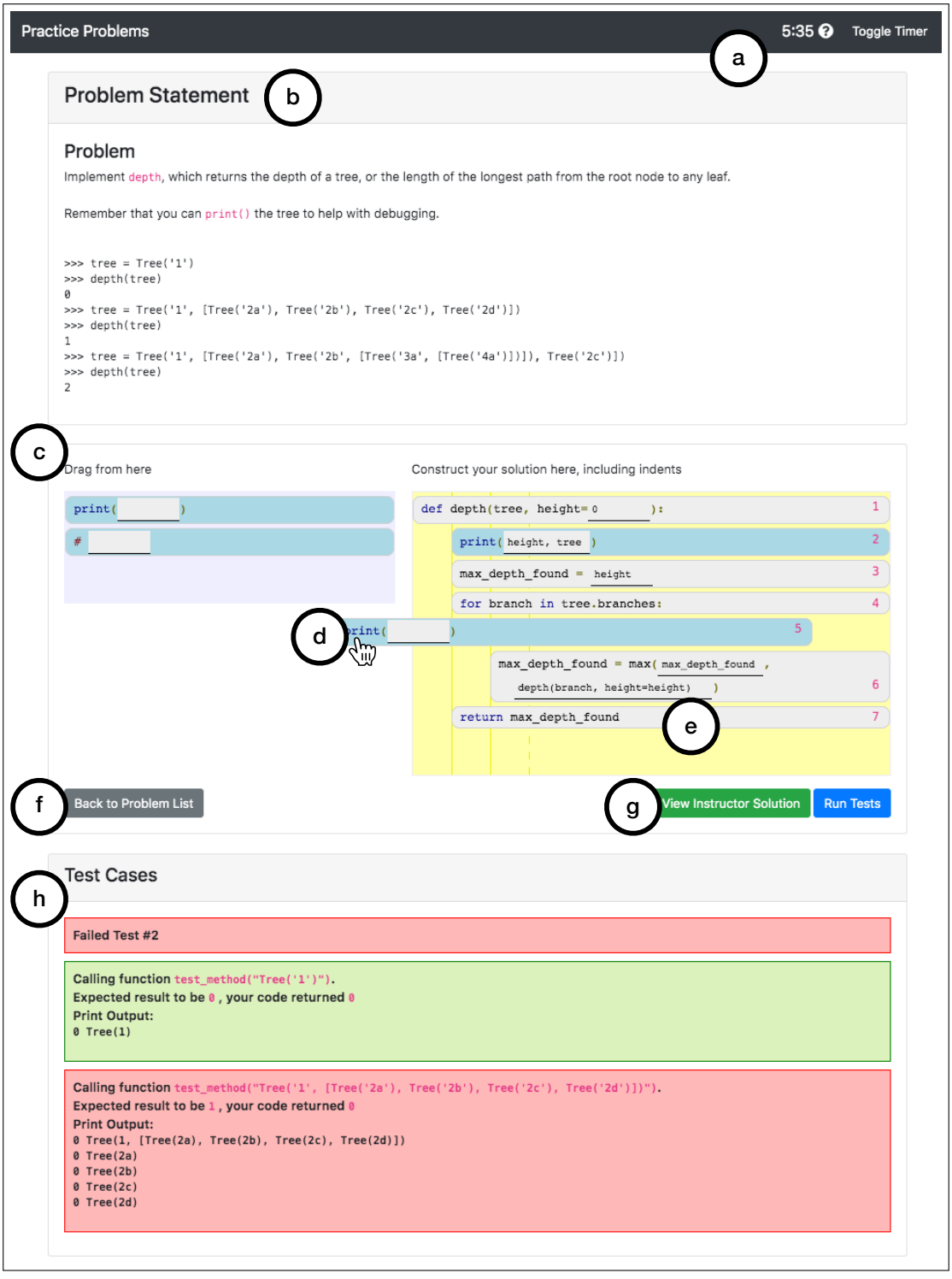

Figure 5: A nearly correct Faded Parsons Problem finding the depth of a Tree. (a) Optional Timer. (b) Problem Description. (c) Faded Parsons Problem interface; participants can drag blocks between the bin (left) and the solution (right). (d) An optional print block being dragged to the right. (e) A blank that has been filled in with code by the student. (f) Students can always navigate back to the exercise list or (g) run tests on their current solution. After effort-completing a exercise, they can view the instructor solution (g). (h) Descriptive test case results up to the first failed test.

research goals to ensure other course content and instruction were not modified.

5.2 Method for Constructing Faded Parsons Problems

Researchers selected which code segments to blank out for Faded Parsons Problems. Required function arguments were never blanked out, unless determining the function arguments was critical to the exercise (e.g., recognizing and using an optional argument). If a variable name was blanked out in its assignment statement, all other references to that variable were also blanked out to avoid confusion in case students selected a different name. Top-level control flow tokens were never blanked out, though conditionals, constants, variable references, and other expressions were sometimes blanked out. The amount of code blanked out from exercises ranged from 0% (only rearranging was required) to 75% of the target code, on average blanking out 32% of the code.

5.3 Study Description

We report on three studies to answer the research questions. A high-level summary is provided in Table 2. Two studies, Study 1 and Study 2, compared the efficacy of three exercise interfaces — `tracing`, `parsons`, and `writing` — in teaching students to acquire programming patterns. A qualitative study was run in-between these two studies to better understand student perception of Faded Parsons Problems.

Figure 6 summarizes the structure of the two studies. In both pattern-focused studies (Study 1 and 2), students knew the concepts (e.g., for loops, Tree structures) necessary to compose the patterns, but the patterns themselves were not explicitly taught as part of the class. These studies began with an `exposure` phase, in which students solved exercises involving different patterns and exercise interfaces. Knowledge Integration [18], an educational framework, suggests that students incrementally acquire generalized knowledge like patterns through new examples, motivating the use of multiple `exposure` exercises for each pattern. The `exposure` phase explores the efficacy of these exercise interfaces in successfully exposing students to relevant patterns. The `exposure` phase is followed by an `acquisition` phase, in which students wrote code in response to a prompt where one of the patterns was applicable. This phase explores if students can express their mastery of patterns through code writing exercises. The first study consisted of additional exercises between the two phases to explore other research questions. There was no explicit indication to students of these different phases, nor did they receive different types of instruction or feedback based on the phase.

In all studies, students worked through exercises in our system in a pre-determined order. Students **effort-completed** a question by correctly solving it and passing all test cases or by both spending at least 10 minutes on the exercise and making 10 consecutively distinct submissions (the course instructor determined this definition for effort-completion). After effort-completing a question, students were able to view an instructor solution or return to the exercise list, after which they could continue on to the next question. Students primarily worked in one of three exercise interfaces, as seen in Figures 2 – 3, 5: code tracing questions (`tracing`), Faded Parsons Problems (`parsons`), and code writing questions (`writing`). Each exercise consisted of an exercise prompt (i.e., problem statement), an interface to work on it, and test cases with their results up to the first failed test case. Problems were designed such that problem statements within a pattern had meaningful differences and were not isomorphic to each other.

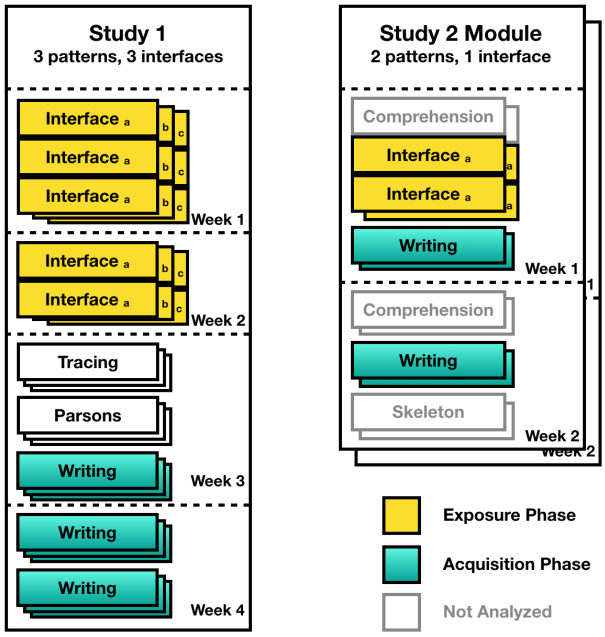

Figure 6: Example sequence of exercises for a single student for Study 1 and a Study 2 module, highlighting a *single* pattern. Each block represents a single exercise. If 2 or more blocks are vertically touching, it means those exercises were consecutive, otherwise some equivalent exercises for other patterns (possibly in different exercise interfaces) occurred between them. The "stacking" of blocks highlights the similar flow for other patterns. That is, in Study 1 Week 2, students worked on 2 exercises for each of 3 (pattern, exercise interface) pairs. Problems in yellow were in the exposure phase, where each pattern was associated with a single exercise interface. Problems in gradient blue were in the acquisition phase, always code writing. Grey exercises were not analyzed in this study. Study 2 consisted of 2 modules, one immediately after the other, represented by the "stacking" of the full study.

5.3.1 Study 1. This study was designed to compare the efficacy of three exercise interfaces — `tracing`, `parsons`, and `writing` — in teaching students programming patterns as well as more general programming ability.

The study consisted of patterns L1, L2, L3 (Table 1). In the `exposure` phase, each pattern was paired with *one* exercise interface such that each exercise interface was used once. For each such pair, five exercises were presented over the course of two weeks. Each student effort-completed a total of 15 exercises in this phase, and all students saw all exercises in the same order with the same

Table 2: High-level attributes of the three reported studies.

Study	Research Questions	Participants	Exercise Interfaces	Duration
Study 1	RQ1, RQ2	50	tracing, parsons, writing	4 weeks
Qualitative Study	RQ3	50	parsons, writing	2 weeks
Study 2	RQ1, RQ2	43	parsons, writing	2 * 2 weeks

exercise interfaces. These exercises measured **Pattern Exposure** and **Active Pattern Exposure**.

In the following week, we instantiated all 9 *(pattern, exercise interface)* combinations: Each student worked on 9 exercises, grouped by exercise interface. The order in which patterns were presented to students was counterbalanced between exercise interfaces. These exercises measured **Actual Difficulty**, as the exercise interfaces were independent of the patterns they were paired with in the exposure phase.

Finally, in addition to the three `writing` exercises from the previous week, students completed 6 more `writing` exercises, 2 for each pattern. These 9 exercises in total represent the acquisition phase, and measured **Pattern Acquisition** and **Code Writing Transfer**. Both metrics are analyzed based on the exercise interface each pattern was paired with in the exposure phase.

All patterns were selected to be of comparable complexity, and the pairing of patterns and exercise interfaces in the learning phase was done randomly after all exercises were created. However, because the study was designed to give a consistent experience to every student and a given pattern was paired with only a single exercise interface in the exposure phase, better performance on that pattern could be due either to that exercise interface being more effective for learning, or to the pattern itself simply being easier to learn.

5.3.2 Study 2. This study was designed to more robustly compare the efficacy of two exercise interfaces — `parsons` and `writing` — in teaching students programming patterns as well as more general programming ability. This study explores more complex patterns (MR1, MR2, OOP1, OOP2) and randomizes interfaces within patterns (unlike Study 1). We do not explore `tracing` exercises in this study due to its poor **Code Writing Transfer** from Study 1.

This study consisted of two modules, one following directly after the other, with students randomly designated into treatment or control. Each module targeted two patterns. The first module covered MR1 and MR2 for all students regardless of treatment, with the second module covering OOP1 and OOP2. Assignment was counterbalanced, so each student was in the control group for one module and the treatment group for the other.

In the exposure phase of each module, patterns were paired with `parsons` for students in treatment or `writing` for students in control. For each pattern, two exercises were presented over the course of one week. Each student effort-completed a total of 4 exercises in this phase.

After the exposure phase, students were given two `writing` exercises for each of the same two patterns, for a total of 4 exercises in this phase.

The modules also consisted of exercises that were not analyzed for this study. First, at the start of every week, students were given a multiple choice comprehension question on the topics covered in lab. These questions were added because the exercises in Study 2 focused on more complex topics. Additionally, each module ended with a code skeleton exercise — effectively a Faded Parsons Problem with the lines already arranged. We hoped the code skeleton exercises would inform us if students were able to demonstrate mastery of the patterns in a more constrained exercise interface. However, upon further analysis, we did not present enough skeleton exercises to analyze.

Study Differences: Below, we emphasize some major differences between the studies that may be helpful in interpreting the results.

- Randomization: In Study 1, all students saw the same materials. In Study 2, students were randomly assigned to treatment or control.
- Dosage: In Study 1, students worked on 5 exercises for each pattern in the exposure phase. In Study 2, they only worked on 2.
- Pattern-Exercise Interface Pairings: In Study 1, each exercise interface was used for a single pattern in the exposure phase, which may have encouraged students to look for similarities. In Study 2, each exercise interface was used for two independent patterns.
- Topics: In Study 1, problem topics changed meaningfully from week to week to match course curriculum, for example focusing on higher-order functions or lambdas. In Study 2, the exercises focused on topics which extended but were supplementary to those in the course curriculum.

5.3.3 Subjective Study. Between the Study 1 and 2, we ran a study focused on gaining preference and self-reported insights from students. This study did not focus on any patterns. In this study, after seeing each problem statement, students were able to choose whether to work on the exercise as `parsons` or `writing`. Additionally, students filled out a survey with questions about their perception of Faded Parsons Problems.

6 RESULTS

The following subsections first describe the statistical measures used and next the data cleaning process for the study. This is followed by analyses of the results corresponding to the major research questions. We report on the degree to which students acquire programming patterns (Section 6.3), what students learn using Faded Parsons Problems beyond learning programming patterns (Section 6.4), and students' subjective responses (Section 6.5). We then synthesize these results (Section 6.6).

6.1 Statistical Measures

Unless otherwise stated, statistical significance is computed as the proportion of relevant submissions matching the measurement in question grouped by student. For Study 1, we analyze the 50 students (17 male, 31 female, 2 unreported) that effort-completed all exercises in the study. We pair by student and use the Friedman test [14] to determine if there is a difference between the three interfaces, then use pairwise Wilcoxon Signed-Rank tests to test for significance. For Study 2, we analyze the 43 students (17 male, 23 female, 3 unreported) that effort-completed all exercises in the study. We use Mann-Whitney U tests to test for significance.

6.2 Data Cleaning

6.2.1 Labeling Pattern Adherence. For all 7 patterns used in this study, researchers created code to automatically detect if a given code submission adhered to the pattern. The code relies on custom string parsing, as submissions might adhere to a pattern even if they cannot be parsed as a valid Abstract Syntax Tree, e.g. if a `:` was missing from a conditional statement. To test the validity of this approach, for each pattern, we randomly selected 25 submissions from the relevant exercises that adhered to the template and 25 submissions that did not. We then removed the algorithm-generated labels, shuffled the results within each pattern, then had two researchers manually annotate the pattern adherence of each submission. We computed Cohen's Kappa score [13] to measure agreement between the annotators and between each of them and the algorithm-generated labels. We found a kappa of 0.89 between the annotators, and an agreement of .94 and .88 respectively between each annotator and the algorithm-generated labels, indicating very good agreement. This gives us confidence that the algorithm can generate labels with comparable accuracy to a human.

6.2.2 Handling Corrupt Data. Researchers examined a sample of the logs and some aggregate data to detect and remove entries where students were abusing the system in some way.

First, because test cases were transparent, some students wrote code to return hard-coded values based on combinations of input arguments, clearly not trying to actually solve the exercise. To address this, researchers added three or more additional test cases to each exercise after the study completed. All submissions were re-tested for correctness based on success on the full set of test cases.

Second, for `parsons` and `writing` exercises, we removed submissions from students if they read and correctly solved the exercise in under 45 seconds. We also excluded submissions from students where their solution exactly matched the instructor solution including the comments explaining the solution.

Finally, from the remaining data to be analyzed, we manually inspected students that did not answer a single `writing` question correctly throughout the study. From this, we remove submissions that appeared designed entirely to get past the effort criteria, such as adding or removing random characters. Across all three criteria, we removed 5 students from analysis.

6.3 Pattern Acquisition

The primary motivation of these studies was to understand how practice with different exercise interfaces affects student acquisition of the programming patterns. For students to learn to recognize and apply patterns, they must first be exposed to the pattern. However, students can correctly solve exercises without ever writing pattern-adherent code or reading the pattern-adherent instructor solution. We first analyze if students are exposed to pattern-adherent code in the exposure phase, either as they construct their own solution or by viewing the instructor solution (Pattern Exposure). We then analyze if students recognize and apply the relevant pattern (Pattern Acquisition) to `writing` exercises based on the exercise interface paired with each pattern in the exposure phase. Results can be seen in Table 3.

6.3.1 Pattern Exposure: Are students exposed to the pattern-adherent code working on an exercise? We posited that one advantage of `parsons` and `tracing` is that they both significantly constrain the solution space, introducing students to particular solutions. Therefore, we would expect that students are more likely to adhere to programming patterns when generating a solution with either interface compared to `writing`. We separately analyze **Active Pattern Exposure**, if students generate pattern-adherent code themselves while completing the exercise, and **Pattern Exposure**, if students ever generate or view pattern-adherent code (e.g., by viewing the instructor solution). We analyze both, because research on Active Learning [24], in which students practice applying skills instead of simply responding to questions, suggests that students will learn patterns better if they construct the patterns themselves.

For `tracing`, by design, Pattern Exposure is 100% and Active Pattern Exposure is 0% as students always view but never construct pattern-adherent code. We find find that `parsons` are more likely than `writing` to expose patterns. The rate of Pattern Exposure is higher for `parsons` than `writing`, 97.2% vs. 39.2% ($p<.001$) in Study 1, 87.8% vs. 64.9% ($p<.001$) in Study 2. The rate of Active Pattern Exposure is also higher for `parsons` than `writing` by a more significant margin, 92.4% vs. 4.4% ($p<.001$) in Study 1, 70.9% vs. 36.9% ($p<.001$) in Study 2.

6.3.2 Pattern Acquisition: Do students recognize and apply the relevant pattern in `writing` *exercises?* Though students are exposed to pattern-adherent code, they may not internalize the patterns as a general solution or may be unable to recall them when given a relevant exercise prompt. Unlike techniques like Pattern-Oriented Instruction [25], students were never given explicit instruction on the patterns or when to use them. We analyze whether students obtain sufficient mastery from the exposure exercises in each exercise interface to both recognize the opportunity to apply a pattern and generate code for that pattern in `writing` exercises in the acquisition phase.

We find that students are more likely to acquire patterns if they are first exposed to them as `parsons` (or `tracing`) compared to `writing`. In Study 1, there is no statistically significant difference in the rate of Pattern Acquisition of `parsons` (55.3%) and `tracing` (44.0%), though both are higher than `writing` (20.7%). In Study 2, the rate of Pattern Acquisition is higher in `parsons` than `writing` (43.3% vs. 33.7%, $p<.01$).

Table 3: Summary statistics related to Pattern Exposure and Acquisition, Code Writing Transfer, and Actual Difficulty. (*) indicates a pairwise-significant difference with one other interface while (**) indicates a pairwise-significant difference with both.

	Study 1			Study 2	
	Tracing	Parsons	Writing	Parsons	Writing
Pattern Exposure	100%*	97.2%*	39.2%**	87.8%*	64.9%*
Active Pattern Exposure	0%*	92.4%**	4.4%*	70.9%*	36.9%*
Pattern Acquisition	44.0%*	55.3%*	20.7%**	43.3%*	33.7%*
Code Writing Transfer	55.3%**	85.3%**	74.7%**	27.7%	28.7%
Actual Difficulty	6.7%**	35.3%*	29.3%*	51.2%	53.5%

6.3.3 A Special Case. Interestingly, we found one pattern, OOP1, where code writing questions were more effective at teaching the programming pattern (though not statistically significant). Further investigation found that for this pattern's `exposure` exercises, Pattern Exposure was nearly identical between the `parsons` condition and the `writing` condition (73.9% vs. 73.7%). However, participants were more likely (not statistically tested) to view the instructor solution in the `writing` condition (57.9% vs. 28.3%).

This pattern involved creating classes with methods that returned generators, and part of the pattern was ensuring that `yield` was at the end of the end of the defining function. This requirement was included as a possibly misguided way to improve readability. The instructor solutions made an explicit reference to this, "# yield is the last line of the loop." This highlights a weakness of `parsons` for certain patterns. Though Active Pattern Exposure might be better in most cases, in this case students were less likely to view the well-documented instructor solutions after solving it correctly in `parsons`, so they did not notice this subtle attribute of the pattern. However, this issue only arose in one of the 7 patterns used in this study. For all exercises, instructors trying to teach patterns could benefit from tools that aggregate student solutions to see if they are being solved in the expected way.

6.4 General Efficacy

We also analyze the effect of introducing Faded Parsons Problems beyond teaching programming patterns. We want to understand how practice with each exercise interface affects students' ability to successfully complete subsequent `writing` exercises with similar solutions (Code Writing Transfer). Open-ended code writing tasks are a well-established measure of mastery in many Computer Science courses. Results can be seen in Table 3.

6.4.1 Code Writing Transfer. Both `parsons` and `tracing` exercises are a meaningfully different type of practice for students compared to `writing`. As `writing` is a well-established goal of programming expertise in introductory Computer Science courses, we evaluate students' success on `writing` questions.

In Study 1, the Code Writing Transfer rate is highest for `parsons` (85.3%), followed by `writing` (74.7%) and then `tracing` (55.3%). However, in Study 2, we find no significant difference between `parsons` (27.7%) and `writing` (28.7%). The poor Code Writing Transfer from `tracing` motivated its exclusion from Study 2.

6.5 Student Perception of Faded Parsons Problems

The Subjective Study was designed to gain insight into student perception of `parsons`. For this, we restrict our analysis to participants that effort-completed all Study 1 exercises, as they had reasonable exposure to both `writing` and `parsons`. All quotes are from open-ended survey responses.

6.5.1 Student Preference for `parsons`. The Subjective Study included 7 exercises where students, after reading the problem statement, could choose to solve an exercise as `parsons` or `writing` and then were asked to explain their choice. We compare to a distribution of students choosing randomly between the two, and find students were much more likely to choose `parsons` (77.6%) over `writing` (22.4%). The primary reason for choosing `parsons` was their perceived difficulty, further analyzed below. Separately, 22 students chose `parsons`in order to focus on the structure of the solution and "allowing them to think like the instructor" (P96), making this the next most frequent explanation. However, 5 students chose `writing` for this same reason, preferring the "additional freedom" (P135) of "starting from scratch" (P114) to create "a more intuitive solution" (P85).

6.5.2 Perceived Difficulty. In the Subjective Study, participants were asked to fill out a survey including a Likert-scale question about whether `writing` was easier to solve, `parsons` was easier to solve, or both are about the same. We find that 26 students (57%) thought `parsons` were easier, 12 students (26%) thought they were similarly difficult, and only 8 students (7%) thought `writing` were easier. Many students echoed this perception in open-ended questions, with 35 explaining their choice of `parsons` as choosing the easier exercise interface. However, 6 chose `writing` for this same reason, thinking they would get "more of a real practice" (P160) when forced "to start thinking on their own" (P137).

6.5.3 Actual Difficulty. We also analyze the rate at which students effort-completed exercises from Study 1 and Study 2 without solving them correctly. Previous work suggests that other variations of Parsons Problems are easier than code writing exercises [12], but Faded Parsons Problems have not yet been assessed for relative difficulty. In Study 1, we analyze exercises from the third week, where we instantiated all 9 *(pattern, exercise interface)* combinations, since the exercise interfaces equally represent each pattern. In Study 2, we analyze the exercises from the exposure phase, comparing treatment to control.

We find that `parsons` and `writing` provide similar difficulty to students. In Study 1 there is no significant difference between `writing` (29.3%) and `parsons` (35.3%). In Study 2, we again find no difference between `writing` (53.5%) and `parsons` (51.2%). These results conflict with the student perception that `parsons` are easier than `writing`.

6.6 Synthesis Of Results

This work has found strong evidence that Faded Parsons Problems are an effective exercise for exposing students to patterns and having them correctly transfer them to subsequent exercises. By contrast, open-ended code writing exercises – a widely used, popular approach to programming exercises – often do not expose students to the intended patterns even when a well-documented instructor solution is provided after the fact. Practice with Faded Parsons Problems also shows similar transfer to general programming success in subsequent code writing exercises over similar content. Though code tracing exercises are fairly effective at having students transfer patterns as well, they are ineffective at transferring to general programming success in subsequent code writing exercises. Code tracing exercises offer clear evidence for how practicing with exercises can support students in practicing certain programming skills while insufficiently practicing others. Faded Parsons Problems appear to offer a good balance between code tracing and code writing exercises, teaching both patterns and general coding skills as well as either of those two interfaces. Furthermore, students had a preference to work with Faded Parsons Problems over code writing exercises, thinking of Faded Parsons Problems as easier problems despite their similar actual difficulty.

7 LIMITATIONS AND FUTURE WORK

Notwithstanding the evidence in favor of integrating Faded Parsons Problems into CS1 courses, several questions remain open. We have not compared Faded Parsons Problems systematically to other Parsons Problem variants or to Code Skeleton questions, in which lines are already arranged but contain blanks, nor have we systematically studied the effects of what and how much code is blanked out in fading the scaffolding. We also do not suggest that all code writing exercises should be replaced with Faded Parsons Problems; indeed, we found one pattern, OOP1, for which code writing exercises provided more effective practice. Finally, we studied a limited number of patterns chosen based on the instructor's existing curriculum rather than on any systematic survey of the importance of different patterns.

The need to be minimally disruptive to the existing curriculum imposed additional constraints that may affect validity of results. First, the study exercises were offered as extra credit rather than required, so self-selection may have biased the study population towards more highly motivated students. Second, Study 2's problems required students to learn more material in addition to the patterns, and students had only 2 exercises in which to learn the patterns, compared to 5 in Study 1. Finally, to remain consistent with the instructor's existing behavior of grouping similar problems together in assignments, both studies intentionally exposed students to the same patterns in consecutive exercises, which might boost students' ability to recognize and apply patterns in the pattern acquisition phase of each study.

On the other hand, the positive results do suggest exploring other uses of scaffolding. For example, instead of giving students instructor solutions after homework assignments were due, what if students instead got points for "unlocking" them by solving Faded Parsons Problems or code tracing questions? In addition, students preferred Faded Parsons Problems over code-writing in part because Faded Parsons Problems were perceived as easier, even though our results suggest otherwise; we have not studied the potentially positive effect of this perception on student self-efficacy.

8 CONCLUSION

The studies we describe provide clear evidence that Faded Parsons Problems are particularly effective at teaching programming patterns in CS1 courses compared to code-tracing and code-writing exercises, without detracting from the ability to *transfer* this knowledge to code-writing exercises. Because Faded Parsons Problems can be created easily from existing code-writing exercises, they can be introduced into existing curricula with minimal disruption. Because they provide immediate feedback, they are particularly valuable for promoting mastery learning in online instruction, where immediate manual feedback may be impossible. Because students piece together and complete an instructor-designed solution, students are more likely to be exposed to a high-quality solution than they would be in constructing their own solution from scratch, which could provide opportunities beyond patterns. These benefits, combined with students' stated preference for Faded Parsons Problems over code-writing exercises, provide strong evidence in favor of integrating such problems widely into introductory programming courses.

ACKNOWLEDGMENTS

We thank Katie Stasaski and Andrew Head for engaging in many fruitful discussions, Michael Ball and the CS88 TAs for supporting of this study in their class, and Danny Chu for assisting in the implementation and analysis the study. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1752814.

REFERENCES

[1] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. 1998. Design patterns: an essential component of CS curricula. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education - SIGCSE '98*. ACM Press, New York, NY, USA, 153–160. https://doi.org/10.1145/273133.273182

[2] David Barr, John Harrison, and Leslie Conery. 2011. Computational Thinking: A Digital Age Skill for Everyone. *Learning and leading with technology* 38 (2011), 20–23.

[3] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 57–73. https://doi.org/10.2190/3lfx-9rrf-67t8-uvk9

[4] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. 2008. Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies* 66, 7 (July 2008), 519–529. https://doi.org/10.1016/j.ijhcs.2007.03.005

[5] Bureau of Labor Statistics 2018-2019. *U.S. Department of Labor, Occupational Outlook Handbook, Software Developers.* Bureau of Labor Statistics. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm (retrieved on April 22, 2020).

[6] William G. Chase and Herbert A. Simon. 1973. Perception in chess. *Cognitive Psychology* 4, 1 (Jan. 1973), 55–81. https://doi.org/10.1016/0010-0285(73)90004-2

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

[8] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and pedagogy. *ACM SIGCSE Bulletin* 31, 1 (March 1999), 37–42. https://doi.org/10.1145/384266.299673

[9] Quintin Cutts, Sarah Esper, Marlena Fecho, Stephen R. Foster, and Beth Simon. 2012. The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes through the Lens of Situated Cognition. In *Proceedings of the ninth annual international conference on International computing education research - ICER '12*. ACM Press, New York, NY, USA, 63–70. https://doi.org/10.1145/2361276.2361290

[10] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a New Exam Question: Parsons Problems. In *Proceeding of the fourth international workshop on Computing education research - ICER '08*. ACM Press, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[11] 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*. ACM, New York, NY, USA, 60–68. https://doi.org/10.1145/3230977.3231000

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

[13] Joseph L Fleiss and Jacob Cohen. 1973. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement* 33, 3 (1973), 613–619.

[14] Milton Friedman. 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the american statistical association* 32, 200 (1937), 675–701.

[15] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*. ACM Press, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

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

[17] Tony Jenkins. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, Vol. 4. Citeseer, Loughborough University, Loughborough, Leicestershire, UK, 53–58.

[18] Marcia C. Linn. 2005. The Knowledge Integration Perspective on Learning and Instruction. In *The Cambridge Handbook of the Learning Sciences*. Cambridge University Press, New York, NY, USA, 243–264. https://doi.org/10.1017/cbo9780511816833.016

[19] Marcia C. Linn and Michael J. Clancy. 1992. The Case for Case Studies of Programming Problems. *Commun. ACM* 35, 3 (March 1992), 121–132. https://doi.org/10.1145/131295.131301

[20] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

[21] Sandra P. Marshall. 1995. *Schemas in Problem Solving*. Cambridge University Press, New York, NY, USA. https://doi.org/10.1017/cbo9780511527890

[22] Tanya J. McGill and Simone E. Volet. 1997. A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education* 29, 3 (March 1997), 276–297. https://doi.org/10.1080/08886504.1997.10782199

[23] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. 2013. Learning computer science concepts with Scratch. *Computer Science Education* 23, 3 (Sept. 2013), 239–264. https://doi.org/10.1080/08993408.2013.832022

[24] Joel Michael. 2006. Where's the evidence that active learning works? *Advances in Physiology Education* 30, 4 (Dec. 2006), 159–167. https://doi.org/10.1152/advan.00053.2006

[25] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '07*. ACM Press, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[26] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 157–163.

[27] David Perkins and Gavriel Salomon. 1992. Transfer Of Learning. *International encyclopedia of education* 2 (1992), 6452–6457.

[28] Mona Rizvi, Thorna Humphries, Debra Major, Meghan Jones, and Heather Lauzun. 2011. A CS0 course using Scratch. *Journal of Computing Sciences in Colleges* 26, 3 (2011), 19–27.

[29] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (June 2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[30] Jorma Sajaniemi and Marja Kuittinen. 2005. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education* 15, 1 (March 2005), 59–82. https://doi.org/10.1080/08993400500056563

[31] C. Scaffidi, M. Shaw, and B. Myers. 2005. Estimating the Numbers of End Users and End User Programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, Hoboken, NJ, USA, 207–214. https://doi.org/10.1109/vlhcc.2005.34

[32] Joshua Shi, Armaan Shah, Garrett Hedman, and Eleanor O'Rourke. 2019. Pyrus: Designing A Collaborative Programming Game to Promote Problem Solving Behaviors. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300886

[33] James C. Spohrer and Elliot Soloway. 1986. Novice mistakes: are the folk wisdoms correct? *Commun. ACM* 29, 7 (July 1986), 624–632. https://doi.org/10.1145/6138.6145

[34] Nathaniel Weinman, Armando Fox, and Marti Hearst. 2020. Exploring Challenging Variations of Parsons Problems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 1349. https://doi.org/10.1145/3328778.3372639

[35] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education* 18, 1 (Dec. 2017), 1–25. https://doi.org/10.1145/3089799

[36] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. 1993. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies* 39, 5 (Nov. 1993), 793–812. https://doi.org/10.1006/imms.1993.1084

[37] 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 (Jan. 2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235

[38] Laura Zavala and Benito Mendoza. 2017. Precursor Skills to Writing Code. *J. Comput. Sci. Coll.* 32, 3 (Jan. 2017), 149–156.